FINDING ECCENTRICITIES IN A TREE

FUNDAMENTALS OF DISTRIBUTED PROCESSING

- PROJECT -

Author: Alexandru Grigoraș

Coordinating teacher: Ș.l.dr.ing. Cristian Bușincu

Contents

1. Description

A common problem in distributed computing is computation in tree networks under the standard restrictions R plus clearly the common knowledge that the network is a tree. Being a tree implies that each entity can determine whether it is a leaf or an internal node.

In this project, the focus is on computing the eccentricities of the nodes. The *eccentricity* of a node x, denoted by $r(x)$, is the largest distance between x and any other node in the tree:

$r(x) = Max\{d(x, y): y \in V\}$ [1]

Over the last seven decades, graph theory has played an increasingly important role in social network analysis. The eccentricity can be used to determine the „most important" or the „central" actor in a social network.

2. Solution

A node x needs to determine the maximum distance from all other nodes in the tree to compute its own *eccentricity,* $r(x)$.

In the first version of this algorithm, the node x broadcasts the request, making itself the root of the tree, and using *convergecast*[1] on this rooted tree, collects the maximum distance to itself. This requires $2(n-1)$ messages, where n is the number of nodes, and is clearly an optimal method, with respect to order of magnitude. To compute the eccentricity of each node in the tree, it would be necessary $2(n-1) * n = 2(n^2 - n)$ messages, which has the complexity $O(n^2)$ and is not optimal.

To produce an optimal solution, $O(n)$, the saturation method is used. The first step is to use the saturation to compute eccentricity of the two saturated nodes and the saturated nodes provides the missing information to other nodes to compute their eccentricity.

The nodes in the saturation method have four states:

$$S = \{available, active, processing, saturated\}$$

Initially, all the entities are available. Arbitrary enitities can start the computation and can be multiple. In the example, the ROOT starts the computation and sends activation messages to others.

The saturation consists of three phases, exemplified in Figure 1:

1) **Activation** phase – is started by the initiators and all the nodes are activated;

   Each initiator sends an activation message to all the neighbors and becomes *active*. Any non-initiator waits for the activation message, becomes *active* and forwards the message to neighbors, excepting the source where the activation came from. After the activation, non-initiators ignore other activation messages. Within finite time, all the nodes become *active*.

   The leaves start the second phase.

[1] Convergecast algorithm is the inverse of a broadcast in a message-passing system, where, instead of a message propagating down from a single root to all nodes, data is collected from outlying nodes through a direct spanning tree to the root.

2) **Saturation** phase – is started by the leaf nodes and couples of neighboring nodes are selected;

The leaves send the message (*M*) to its only neighbor (selected as „parent") and becomes *processing*. An internal node waits until it has received a message *M* from all it's neighbors *but one*, sends a message *M* to that neighbor that will be considered its „parent" and becomes *processing*.

If a processing node receives a message from its parent, it becomes *saturated*.

3) **Resolution** phase – started by the selected pairs and every node calculates their eccentricity.

It is started by the saturated nodes that already have all the data received from the other nodes to calculate their eccentricity.

The goal is to have all the nodes determining their eccentricity. The information available at each enity at the end of the saturation stage is *almost* sufficient to make them compute their eccentricity. Considering an entity u that sends a message to its parent v, it already knows the maximum distance from all the entities *except* the one in the tree $T(v-u)$. The only missing information $d[u,v] = Max\{d(u,y): y \in T[v-u]\}$. If the parent isn't saturated, it also doesn't have the missing information. Only the saturated nodes can send that information.

Thus, the resolution stage is used to provide missing information. Starting from the saturated nodes, once the node receives the missing information from a neighbor, it computes its eccentricity and provides the missing information to its neighbors.

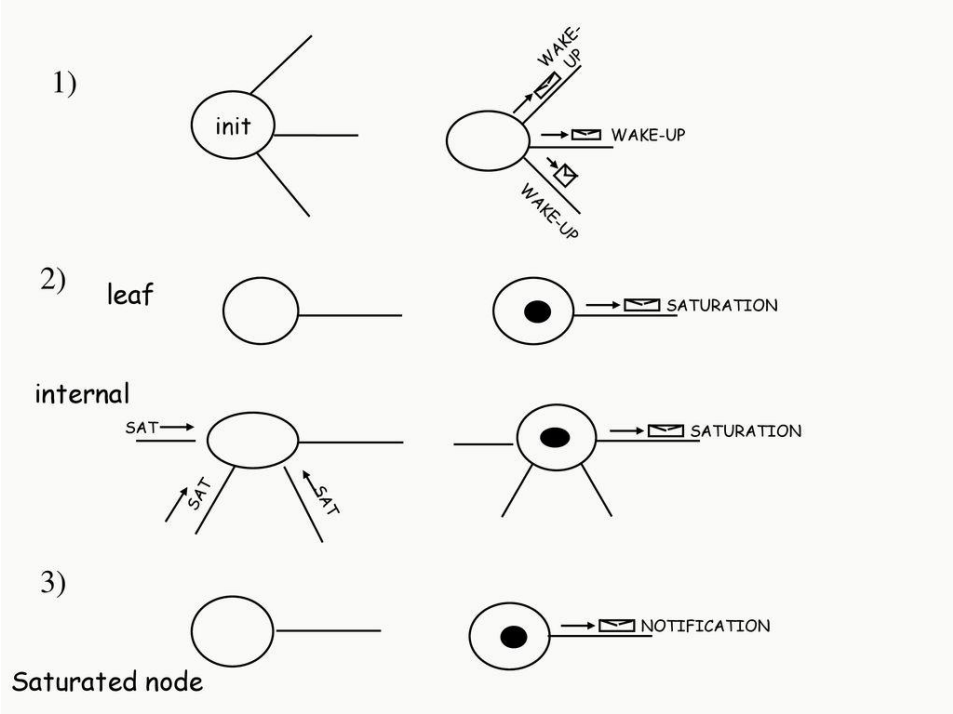


Figure 1. The saturation method

3. Algorithm

3.1. Description

Each node in the tree knows its neighbors and if its a leaf or an internal node. In the beginning, all processing nodes are AVAILABLE.

The nodes can be in the following states, explained in Chapter 2:

- *AVAILABLE* – the state of the nodes in the beginning;
- *ACTIVE* – if a node is not saturated;
- *PROCESSING* – if a node is saturated;
- *SATURATION* – if a node is saturated and start the resolution phase;
- *DONE* – terminal status of the node where it finishes its excution.

Send and receive are functions from MPI. Each node has a vector where each location stores the distance from each neighbor. Procedure *Initialize* sets the values of distance vector to 0, *Prepare_Message* creates the message M to be sent (the maximum distance known + 1), *Process_Message* add the received message to the distance vector and *Calculate_Eccentricity* determines the node excentricity by getting the largest distance known. *Resolve* is started by the saturated nodes and sent recursively in the tree to all other nodes, except parent. It processes the received message, calculates the eccentricity and forwards the resolution to neighbors, except parent, with maximum distance for each node.

When they finish the resolution, the nodes enter in the DONE state and finish their execution.

3.2. Pseudocode

Finding excentricities:

- Status: $S = \{\text{AVAILABLE}, \text{ACTIVE}, \text{PROCESSING}, \text{SATURATION}, \text{DONE}\}$; $S_{INIT} = \{\text{AVAILABLE}\}$;
- Restrictions: $R \cup T$.

Algorithm 1: Finding Eccentricities in a Tree

```
AVAILABLE
    Spontaneously
    begin
        send(Activate) to N(x);
        Initialize;
        Neighbors:= N(x);
        if |Neighbors|=1 then
            Prepare_Message;
            parent ⇐ Neighbors;
            send(M) to parent;
            become PROCESSING;
        else become ACTIVE;
        endif
```

```
Procedure Initialize
    begin
        Distance[x]:= 0;
    end

Procedure Prepare_ Message
    begin
        maxdist:= 1+ Max{Distance[*]};
        M:=("Saturation", maxdist);
    end

Procedure Resolve
    begin
```

```
    end

    Receiving(Activate)
    begin
        send(Activate) to N(x)−{sender};
        Initialize;
        Neighbors:= N(x);
        if |Neighbors|=1 then
            Prepare_Message;
            parent ⇐ Neighbors;
            send(M) to parent;
            become PROCESSING;
        else become ACTIVE;
        endif
    end

ACTIVE
    Receiving(M)
    begin
        Process_Message;
        Neighbors:= Neighbors−{sender};
        if |Neighbors|=1 then
            Prepare_Message;
            parent ⇐ Neighbors;
            send(M) to parent;
            become PROCESSING;
        endif
    end

PROCESSING
    Receiving(„Resolution", dist)
    begin
        Resolve;
    end
```

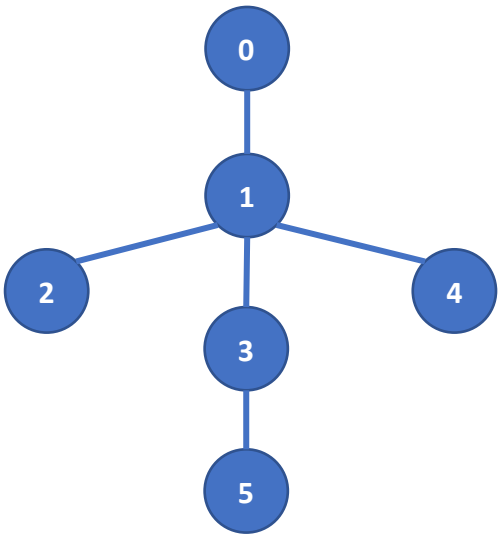
```
        Process_Message;
        Calculate_Eccentricity;
        forall y ∈ N(x)−{parent} do
            maxdist:= 1+ Max{Distance[z]: z
                    ∈ N(x)−{parent,y}};
            send("Resolution", maxdist) to y;
        endfor
        become DONE;
    end

Procedure Process_Message
    begin
        Distance[sender]:= Received_distance;
    end

Procedure Calculate_Eccentricity
    begin
        r(x):= Max{Distance[z]: z ∈ N(x)};
    end
```

3.3. Example

Let T be a tree [2].



Given the above tree, the eccentricities of vertices are:

r(0) = 3	r(1) = 2	r(2) = 3
r(3) = 2	r(4) = 3	r(5) = 3

3.4. Demonstration

Lemma: Exactly two processing nodes will become saturated; furthermore, these two nodes are neighbors and are each other's parent.

Demonstration: An entity sends a message *M* only to its parent and becomes saturated after receiving a message *M* from its parent. For example, an arbitrary node *u* is choosed and traverse the „up" edge of *x* (from parent). By moving the „up" edge, a *saturated* node s_1 is met (since there are no cycles in the tree). This node has become *saturated* when receiving a message *M* from its parent s_2. Since s_2 has sent a message *M* to s_1, this implies that s_2 must have been *processing* and must have considered s_1 as its parent. When the message *M* from s_1 will arrive at s_2, s_2 will also become saturated. Thus, there exist at least two nodes that become *saturated* and are each other's parent. Assuming that there are more than two saturated nodes, then there exist two saturated nodes, x and y, such that d(x,y)≥2 (d is the distance function). Consider a node z on the path from x to y; z could not send a message *M* toward both x and y; therefore, one of the nodes cannot be saturated. Therefore, the lemma holds.

Important: It depends on the communication delays which entities will become saturated and it is therefore totally unpredictable. Subsequent executions with the same initiators might generate different results. *Any pair of neighbors could become saturated.*

The correctness follows from the fact that there are at least two saturated nodes that know their eccentricity and send information to others to computer theirs.

3.5. Complexity analysis

Message Complexity:

The complexity of the phases are:

- Activation –> $n + k^* - 2$ mesages (k^* initiators)
- Saturation –> n messages
- Resolution –> $n - 2$ messages

In total there are $3n + k^* - 2$ messages and the complexity is:

$$M[Eccentricities] = O(n)$$

Time complexity:

First, the time complexity of the saturation is calculated

I ⊆ V –> the set of initiator nodes;

L ⊆ V –> the set of leaf nodes;

t(x) –> the time delay, from the initiation of the algorithm, until node x becomes active.

To become *saturated*, node *s* must have waited until all the leafs have become *active* and the *M* messages originated from them have reached *s*. It must have waited Max{t(l)+d(l,s):l ∈ L}. To become *active*, a non-initiator node *x* must have waited for an "Activation" message to reach it, while there is no additional waiting time for an initiator node; thus, t(x)=Min{d(x,y)+ t(y):y ∈ I}. Therefore, the total delay, from the initiation of the algorithm, until *s* becomes saturated is:

$$T[Full\ Saturation] = Max\{Min\{d(l,y) + t(y)\} + d(l,y): y \in I, l \in L\}.$$

The time costs of finding the eccentricities will be the one experienced by Full Saturation plus the ones required by the resolution phase where non-saturated nodes receive the missing information (Sat denotes the set of the two saturated nodes):

$$T[Eccentricities] = T[Full\ Saturation] + Max\{d(s,x): s \in Sat, x \in V\}.$$

3.6. Implementation

Implementation is done using C programming language with MPI (Message Passing Interface).

The algorithm is presented below:

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

// Macros
#define ROOT 0
#define NR_NODES 6
#define TRUE 1
```

```
#define FALSE 0

// The status of the node
enum STATUS_ENUM {
    AVAILABLE = 0,
    ACTIVE = 1,
    PROCESSING = 2,
    DONE = 3
};

// The type of the message that is sent to neighbors
enum MESSAGE_TYPE {
    ACTIVATE = 0,
    SATURATION = 1,
    RESOLUTION = 2
};

// Initialization of the distances vector with 0 on all locations
void Initialize(int * dist, int np) {
  for (int i = 0; i < np; i++) {
    dist[i] = 0;
  }
}

// Build the message value with the maximum distance from neighbors + 1
int Prepare_Message(int * dist, int np) {
  int i;
  int maxdist = dist[0];

  for (i = 1; i < np; i++) {
    if (dist[i] > maxdist) {
      maxdist = dist[i];
    }
  }
  maxdist++;
  return maxdist;
}

// Update the distances vector with the received distance from sender
void Process_Message(int * dist, int received_distance, int sender) {
  dist[sender] = received_distance;
}

// Calculate the node eccentricity
int Calculate_Eccentricities(int * dist, int np) {
```

```
  int i;
  int maxdist = dist[0];
  for (i = 1; i < np; i++) {
    if (dist[i] > maxdist) {
      maxdist = dist[i];
    }
  }
  return maxdist;
}

// Saturated nodes enter in the resolution stage:
// - calculate the eccentricity
// - send the maxdist accordingly to each neighbor
// - return the eccentricity
int Resolve(int * dist, int nodes[][NR_NODES], int received_distance, int
my_rank, int parent, int sender, int np) {
  int dest, tag;
  int maxdist;
  int message;
  int eccentricity;

  Process_Message(dist, received_distance, sender);
  eccentricity = Calculate_Eccentricities(dist, np);

  for (dest = 0; dest < np; dest++) {
    if (nodes[my_rank][dest] != 0 && dest != parent) {
      maxdist = 0;
      for (int i = 0; i < np; i++) {
        if ((dist[i] > maxdist) && (i != dest)) {
          maxdist = dist[i];
        }
      }
      tag = RESOLUTION;
      message = maxdist + 1;
      MPI_Send( & message, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
    }
  }
  return eccentricity;
}

// Display a vector to console
void Print_vector(int * v, int nr) {
  printf("V = [ ");
  for (int i = 0; i < nr; i++) {
    printf("%d ", v[i]);
```

```
  }
  printf("]\n");
}

// Main function for all the processes
int main(int argc, char * argv[]) {
  int my_rank;          // rank of process
  int nr_processes;     // number of processes
  int source;           // rank of sender
  int dest;             // rank of receiver
  int tag = 0;          // tag for messages
  int message = 0;      // storage for message
  MPI_Status status;    // return status for receive
  int i;                // index
  int node_status = AVAILABLE; // status of the node; initially is set to
AVAILABLE
  int parent = -1; // parent of the node; initially the parent is unknown
  int nr_neighbors = 0; // the number of neighbors
  int temp_nr_neighbors = 0; // temporary number of neighbors
  int nodes[NR_NODES][NR_NODES] =  // adjacency matrix of the tree
  {
    { 0,1,0,0,0,0 },
    { 1,0,1,1,1,0 },
    { 0,1,0,0,0,0 },
    { 0,1,0,0,0,1 },
    { 0,1,0,0,0,0 },
    { 0,0,0,1,0,0 }
  };

  int distances[NR_NODES];     // distance vector
  int finished = FALSE;        // finish flag
  int eccentricity = -1;       // calculated eccentricity

  // start up MPI
  MPI_Init( & argc, & argv);
  // find out process rank
  MPI_Comm_rank(MPI_COMM_WORLD, & my_rank);
  // find out number of processes
  MPI_Comm_size(MPI_COMM_WORLD, & nr_processes);

  // getting the number of neighbors for each node
  for (i = 0; i < nr_processes; i++) {
    if (nodes[my_rank][i]) {
      nr_neighbors++;
    }
```

```
  }
  // and store it to a temporary variable
  temp_nr_neighbors = nr_neighbors;

  // main loop where each process goes through the available states
  while (!finished) {
    switch (node_status) {
      // ACTIVATION state:
      // - the root is sending activation messages to its neighbors
      // - other nodes forward the activation to their neighbors
      //- the root chooses the parent the first neighbor
      //- other nodes choose the parent the neighbor if leaf or the last
neighbor from which it received a message
    case AVAILABLE:
      if (my_rank == ROOT) {
        // sending activation message to neighbors
        for (dest = 0; dest < nr_processes; dest++) {
          if (nodes[my_rank][dest]) {
            tag = ACTIVATE;
            message = 0;
            MPI_Send( & message, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
          }
        }

        // initialize and send SATURATION to parent and set status to
PROCESSING if process is leaf
        // otherwise set status to active
        Initialize(distances, nr_processes);
        if (nr_neighbors == 1) {
          for (dest = 0; dest < nr_processes; dest++) {
            if (nodes[my_rank][dest]) {
              parent = dest;
            }
          }
          message = Prepare_Message(distances, nr_processes);
          tag = SATURATION;
          dest = parent;
          MPI_Send( & message, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
          node_status = PROCESSING;
        } else {
          node_status = ACTIVE;
        }
      } else {
        // receiving activation message
```

```
        MPI_Recv( & message, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, & status);
        source = status.MPI_SOURCE;
        tag = status.MPI_TAG;

        // forward the activation
        if (tag == ACTIVATE) {
          // sending activation message to neighbors, except source
          for (dest = 0; dest < nr_processes; dest++) {
            if (nodes[my_rank][dest]) {
              if (dest != source) {
                tag = ACTIVATE;
                message = 0;
                MPI_Send( & message, 1, MPI_INT, dest, tag,
MPI_COMM_WORLD);
              }
            }
          }

          // initialize and send SATURATION to parent and set status to
PROCESSING if process is leaf
          // otherwise set status to active
          Initialize(distances, nr_processes);
          if (nr_neighbors == 1) {
            parent = source;
            tag = SATURATION;
            message = Prepare_Message(distances, nr_processes);
            dest = parent;
            MPI_Send( & message, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
            node_status = PROCESSING;
          } else {
            node_status = ACTIVE;
          }
        }
      }
      break;

      // ACTIVE STAGE:
      // - process incoming messages
      // - on the last message received send SATURATION message to source
and become PROCESSING
    case ACTIVE:
      MPI_Recv( & message, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, & status);
      source = status.MPI_SOURCE;
```

```
      tag = status.MPI_TAG;
      temp_nr_neighbors -= 1;
      Process_Message(distances, message, source);
      if (temp_nr_neighbors == 1) {
        message = Prepare_Message(distances, nr_processes);
        parent = source;
        tag = SATURATION;
        dest = parent;
        MPI_Send( & message, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
        node_status = PROCESSING;
      }
      break;

      // PROCESSING STAGE:
      // - saturated nodes are starting the resolution step where they
send the needing information to other nodes
    case PROCESSING:
      MPI_Recv( & message, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, & status);
      source = status.MPI_SOURCE;
      tag = status.MPI_TAG;

      if (tag == SATURATION) {
        eccentricity = Resolve(distances, nodes, message, my_rank,
parent, source, nr_processes);
        tag = RESOLUTION;
        dest = parent;
        message = eccentricity;
        MPI_Send( & message, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
        node_status = DONE;
      } else if (tag == RESOLUTION) {
        eccentricity = Resolve(distances, nodes, message, my_rank,
parent, source, nr_processes);
        node_status = DONE;
      }
      break;

      // DONE State
      // - nodes finish their execution
    case DONE:
      printf("r(%d) = %d\n", my_rank, eccentricity);
      fflush(stdout);
      finished = TRUE;
      break;
    }
```

```
    }

    // shut down MPI
    MPI_Finalize();

    return 0;
}
```

3.7. Testing

The testing is done with logging data to console. The state each node ensures that the algorithm functions properly. To be noted that different executions can cause different results because of the delays of messages sent and task managing of the processes from the host operating system.

Below are the logs from the algorithm execution with the tree from chapter 3.3:

```
#PTP job_id=2764
[0] AVAILABLE and sending ACTIVATION to neighbors
[0] AVAILABLE and sending SATURATION to 1
[1] AVAILABLE and receiving ACTIVATION from 0
[2] AVAILABLE and receiving ACTIVATION from 1
[2] AVAILABLE and sending SATURATION to 1
[2] PROCESSING from 1 and sending RESOLUTION to neighbors
r(2) = 3
[3] AVAILABLE and receiving ACTIVATION from 1
[3] ACTIVE and receiving SATURATION from 1
[3] ACTIVE and receiving SATURATION from 5
[3] ACTIVE and sending SATURATION to 5
[4] AVAILABLE and receiving ACTIVATION from 1
[4] AVAILABLE and sending SATURATION to 1
[4] SATURATED from 1 and sending RESOLUTION to parent
r(4) = 2
[5] AVAILABLE and receiving ACTIVATION from 3
[5] AVAILABLE and sending SATURATION to 3
[1] ACTIVE and receiving SATURATION from 0
[1] ACTIVE and receiving SATURATION from 2
[1] ACTIVE and receiving SATURATION from 4
[1] ACTIVE and sending SATURATION to 4
[1] PROCESSING from 4 and sending RESOLUTION to neighbors
r(1) = 2
[0] PROCESSING from 1 and sending RESOLUTION to neighbors
r(0) = 3
[5] SATURATED from 3 and sending RESOLUTION to parent
r(5) = 2
```

```
[3] PROCESSING from 5 and sending RESOLUTION to neighbors
r(3) = 2
```

4. Conclusions

In the last decades, graph theory has solved numerous real-world problems. One of them is the emerging social network that can be modelled using graphs. With a numerous number of users, its important to know the ones that can influence others. A simple method to solve this problem is to determine the connections from other users and get the most important ones. Because the number of users is very large, a distributed algorithm is necessary for solving this problem in a short period of time.

In this project was proved that finding the eccentricities of the nodes in a tree can be done using a distributed method that can be executed in parallel. The algorithm is efficient in messages exchanged between nodes and has a high speed of execution.

5. References

[1] N. Santoro, Design and Analysis of Distributed Algorithms, Ottawa: WILEY-INTERSCIENCE, 2006.

[2] I. Baidari, R. Roogi și S. Shinde, „Algorithmic Approach to Eccentricities, Diameters and Radii of Graphs using DFS," *International Journal of Computer Applications 54(18):1-4,* vol. 54, pp. 1-4, 2012.