Technical University of Cluj-Napoca
Computer Science Department

Distributed Systems

- Master in CS -

C5
Basic Distributed Algorithms

Fall 2017

Introduction
Program structure

- For algorithm description will use Hoare's CSP (Concurrent Sequential Processes) language for the efficient synchronizing of communicating processes
- The most general control structure for any process, part of a distributed application is based on the CSP repetitive command over the alternative command on multiple guarded commands:

$$* [G_1 \rightarrow CL_1 \,||\, G_2 \rightarrow CL_2 \,||\, \ldots \,||\, G_k \rightarrow CL_k]$$

Introduction
Program structure (2)

$$*[\ G_1 \rightarrow CL_1 \ ||\ G_2 \rightarrow CL_2 \ ||\ \ldots \ ||\ G_k \rightarrow CL_k\ ]$$

- Repetitive command (an infinite loop)
  - Denoted by *
  - Inside the repetitive command - alternative command over guarded commands
- The alternative command
  - denoted by a sequence of || separating guarded commands
  - Specifies execution of exactly one of its constituent guarded commands
- The guarded command

  $$G \rightarrow CL$$

  - The guard G - boolean expression
  - CL - list of commands that are only executed if G is true
- The guard expression may contain a term to check if a message from a/any other process has arrived
- If more than one guard is true
  - one of those successful guarded commands is non-deterministically chosen for execution
- When a guarded command $G_m \rightarrow CL_m$ comes to execution
  - CL_m execution is atomic with the execution of G_m
- The alternative command over the guarded commands fails if all the guards fail

Introduction
Program structure (4)

- The structure of distributed programs (see next slide)
  - Similar semantics to that of CSP although the syntax has evolved to something very different

Introduction
Program structure (5)

1. Declare process-local variables whose scope is global to the process, and message types
2. Declare shared variables, if any, (for distributed shared memory systems) . They are explicitly labeled as such
3. Initialization code.
4. The **repetitive** and the **alternative** commands are not explicitly shown as in the presented syntax
5. The **guarded** commands are shown as explicit modules or procedures (e.g. lines 1–4 in the Algorithm) The guard usually checks for the arrival of a message of a certain type, perhaps with additional conditions on some parameter values and other local variables
6. The body of the procedure gives the list of commands to be executed if the guard evaluates to true
7. Process termination may be explicitly stated in the body of any procedure(s)
8. The symbol $\perp$ is used to denote an undefined value. When used in a comparison, its value is $-\infty$.

```
(local variables)
int parent ⟵ ⊥
set of int Children, Unrelated ⟵ ∅
set of int Neighbors ⟵ set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ⟵ i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ⟵ j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ⟵ Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ⟵ Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

Introduction
Elementary DS related graph algorithms (1)

- Objective
  - Presentation of the elementary distributed algorithms on graphs
- Assumptions
  - Un-weighted undirected edges
  - Asynchronous execution by the processors (unless otherwise specified)
  - Communication by message-passing on the edges
- Facts of Distributed Algorithms
  - Each node has only partial view of the graph (system) (The set of its immediate neighbors)
  - A node can communicate with only its immediate neighbors along the incident edges

Introduction

Elementary DS related graph algorithms (1)

- Building DS overlay spanning trees
- Spanning trees - efficient ways of information distribution and collection in DS
- Building spanning tree algorithms
  - Synchronous algorithms
  - Asynchronous algorithms

1. Synchronous single-initiator spanning tree algorithm using flooding

- The code for all processes
  - Symmetrical
  - Proceeds in rounds
- Assumption
  - Algorithm assumes a designated root node (**root**), which initiates the algorithm
- The pseudo-code for each process $\mathbf{P_i}$ is shown in Algorithm 1
  - The **root** initiates a flooding of QUERY messages in the graph to identify tree edges
  - The parent of a node is that node from which a QUERY is first received
  - If multiple QUERYs are received in the same round, one of the senders is randomly chosen as the parent
  - Graph **diameter** – the largest number of edges that need to be traversed from the root to a graph vertex
- Exercise
  - Modify the algorithm so that each node identifies not only its parent node but also all its children nodes

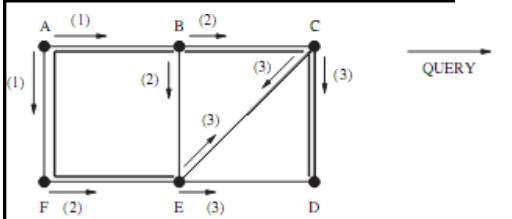
Synchronous single-initiator spanning tree algorithm using flooding

```
(local variables)
int visited, depth ← 0
int parent ← ⊥
set of int Neighbors ← set of neighbors
(message types)
QUERY

(1)  if i = root then
(2)      visited ← 1;
(3)      depth ← 0;
(4)      send QUERY to Neighbors;
(5)  for round = 1 to diameter do
(6)      if visited = 0 then
(7)          if any QUERY messages arrive then
(8)              parent ← randomly select a node from which
                     QUERY was received;
(9)              visited ← 1;
(10)             depth ← round;
(11)             send QUERY to Neighbors \ {senders of
                     QUERYs received in this round};
(12)     delete any QUERY messages that arrived in this round.
```

Algorithm 1 (source [3]) - Spanning tree algorithm: the synchronous breadth-first search (BFS) spanning tree algorithm. The code is shown for processor Pi where $1 \leq i \leq n$

Example



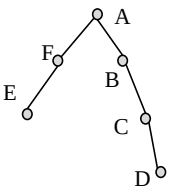


Figure 1 (source [3]) Synchronous single-initiator example

- Figure 1 - Example algorithm execution
- Node A as initiator
- The resulting tree
  - Is shown in boldface
  - The round numbers in which the QUERY messages are sent are indicated next to the messages
- For example
  - At the end of round 2, E receives a QUERY from B and F
  - E randomly chooses F as the parent
- A total of nine QUERY messages are sent in the network which has eight links

Termination

- The algorithm terminates after all the rounds are executed
- Algorithm can be straightforward modified so that a process exits after the round in which it sets its parent variable

Complexity

- The local space complexity at a node
  - of the order of the degree of edge incidence
- The local time complexity at a node
  - of the order of (diameter + degree of edge incidence)
- The global space complexity
  - the sum of the local space complexities
- This algorithm sends at least one message per edge, and at most two messages per edge.
  - The number of messages is between **l** and 2**l**.
- The message time complexity is
  - **d** rounds or message hops

Other features

- The resulted spanning tree
  - Breadth-first tree (BFS)
- The code is the same for all processes but the pre-designated root executes a different logic.

=> In the strictest sense, the algorithm is asymmetric

2. Asynchronous single-initiator spanning tree algorithm using flooding

- Assumption
  - This algorithm assumes a designated root node which initiates the algorithm
- The pseudo-code for each process P_i is shown in Algorithm 2
- Messages: QUERY, ACCEPT, REJECT
- The root initiates a flooding of QUERY messages in the graph to identify tree edges
- The parent of a node is that node from which a QUERY is first received
  - an ACCEPT message is sent in response to such a QUERY
  - Other QUERY messages received are replied to by a REJECT message
- Algorithm termination (at each node)
  - Each node terminates its algorithm when it has received from all its non-parent neighbors a response to the QUERY sent to them
- Procedures 1, 2, 3, and 4 are each executed atomically

Asynchronous single-initiator spanning tree algorithm using flooding

```
(local variables)
int parent ← ⊥
set of int Children, Unrelated ← ∅
set of int Neighbors ← set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ← i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ← j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ← Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ← Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

Algorithm 2 (source [3]) - Spanning tree algorithm: the asynchronous algorithm assuming a designated root that initiates a flooding. The code is shown for processor Pi where $1 \leq i \leq n$

Asynchronous single-initiator spanning tree algorithm using flooding

- This is an asynchronous system
  - No bound on the time it takes to propagate a message

  => no notion of a message round
  - Each node needs to track its neighbors to determine which nodes are its children set and which nodes are not (unlike in the synchronous algorithm)
    - » This tracking is necessary in order to know when to terminate

Asynchronous single-initiator spanning tree algorithm using flooding

- To distinguish between the child nodes and non-child nodes, the algorithm uses two messages types (besides the QUERY):
  - ACCEPT (+ ack) and
  - REJECT (- ack)
- After sending QUERY messages on the outgoing links, the sender needs to know how long to keep waiting
  - This is accomplished by requiring each node to return an "acknowledgement" for each QUERY it receives
  - The acknowledgement message has to be of a different type than the QUERY type

Asynchronous single-initiator spanning tree algorithm using flooding **Termination**

- The termination condition is formulated as a set equality:

  Children **U** Unrelated == Neighbors \ { parent }
- Notes on distributed algorithms termination
  - For some algorithms (such as this algorithm) it is possible to locally determine the termination condition
  - For other algorithms, the termination condition is not locally determinable
    - » An explicit termination detection algorithm needs to be executed

Asynchronous single-initiator spanning tree algorithm using flooding **Complexity**

- The local space complexity at a node
  - of the order of the degree of edge incidence
- The local time complexity at a node
  - of the order of the degree of edge incidence
- The global space complexity
  - the sum of the local space complexities
- Message complexity
  - The algorithm sends
    - » at least two messages (QUERY and its response) per edge, and
    - » at most four messages per edge (when two QUERIES are sent concurrently, each will have a REJECT response).

  => The number of messages is between 2**l** and 4**l**
  - The message time complexity
    - » Assuming synchronous communication

    => There are (d+1) message hops
    - » In an asynchronous system
      - no claim can be made about the tree obtained
      - its depth may be equal to the length of the longest path from the root to any other node, which is bounded only by n−1 corresponding to a depth-first tree

Asynchronous single-initiator spanning tree algorithm using flooding **Example**

- Figure 2
  - Example execution of the asynchronous algorithm (i.e., in an asynchronous system)
  - The resulting spanning tree rooted at A is shown in boldface
  - The numbers next to the QUERY messages indicate the approximate chronological order in which messages get sent.
- Each procedure is executed atomically:

  => the sending of a message sent at a particular time is triggered by the receipt of a corresponding message at the same time
- Concurrently and independently actions
  - Indicated by the same numbers used for messages sent by different nodes
- ACCEPT and REJECT messages
  - not shown to keep the figure simple
- It does not matter when the ACCEPT and REJECT messages are delivered

Asynchronous single-initiator spanning tree algorithm using flooding
Example

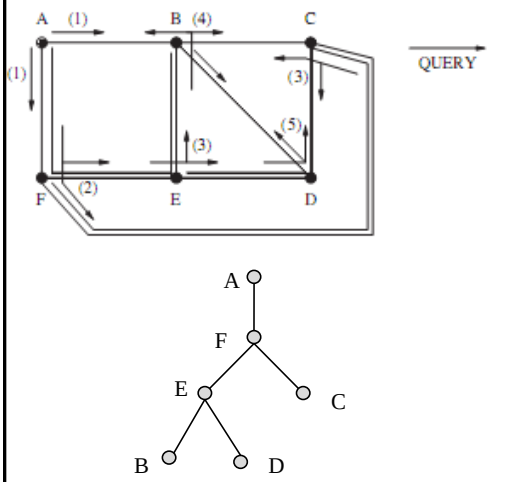


Figure 2 (source [3]) Asynchronous single-initiator example

Asynchronous single-initiator spanning tree algorithm using flooding
Example

1. A sends a QUERY to B and F.
2. F receives QUERY from A and determines that AF is a tree edge.
   F forwards the QUERY to E and C.
3. E receives a QUERY from F and determines that FE is a tree edge.
   E forwards the QUERY to B and D.
   C receives a QUERY from F and determines that FC is a tree edge.
   C forwards the QUERY to B and D.
4. B receives a QUERY from E and determines that EB is a tree edge.
   B forwards the QUERY to A, C, and D.
5. D receives a QUERY from E and determines that ED is a tree edge.
   D forwards the QUERY to B and C.

Asynchronous single-initiator spanning tree algorithm using flooding
Example

- Each node sends an ACCEPT message (not shown in Figure 2 for simplicity) back to the parent node from which it received its first QUERY
- ACCEPT message enables the parent (i.e., the sender of the QUERY)
  - to recognize that the edge is a tree edge, and
  - to identify its child
- All other QUERY messages are negatively acknowledged by a REJECT (also not shown for simplicity)
  - A REJECT gets sent on each back edge (such as BA) and each cross edge (such as BD, BC, and CD) to enable the sender of the QUERY on that edge to recognize that that edge does not lead to a child node

Asynchronous single-initiator spanning tree algorithm using flooding
Example

- On each tree edge
  - two messages (a QUERY and an ACCEPT) get sent
- On each cross-edge and each back-edge
  - four messages (two QUERY and two REJECT) get sent
- **Note**. This algorithm does not guarantee a breadth-first tree.
- **Exercise proposal** - modify this algorithm to obtain a BFT tree

3. Asynchronous concurrent-initiator spanning tree algorithm using flooding

- Concurrent initiation assumption
  - Any node may spontaneously initiate the spanning tree algorithm
  - Initiation precondition - the initiator has not already been invoked locally due to the receipt of a QUERY message
    - » In other words: two or more processes that are not yet participating in the algorithm initiate the algorithm concurrently
- Modified Algorithm 2 => Algorithm 3
  - Algorithm 3 objective is to construct a single spanning tree
- When concurrent initiations are detected, two options are available (see Option 1 and Option 2 in the following slides)
- **Note**.
  - Even though there can be multiple concurrent initiations, along any single edge, only two concurrent initiations will be detected

Asynchronous concurrent-initiator spanning tree algorithm using flooding

- **Option 1**
  - When two concurrent initiations are detected by two adjacent nodes that have sent to each other a QUERY from different initiations, the two partially computed spanning trees can be merged
    - » This merging cannot be done based only on local knowledge or there might be cycles

Asynchronous concurrent-initiator spanning tree algorithm using flooding **Example**

- In Figure 3, consider that the algorithm is initiated concurrently by A, G, and J.
- The dotted lines show the portions of the graphs covered by the three algorithms.
- At this time, the initiations by A and G are detected along edge BD, the initiations by A and J are detected along edge CF, the initiations by G and J are detected along edge HI.
- If the three partially computed spanning trees are merged along BD, CF, and HI, there is no longer a spanning tree
- Even if there are just two initiations, the two partially computed trees may "meet" along multiple edges in the graph, and care must be taken not to introduce cycles during the merger of the trees

Asynchronous concurrent-initiator spanning tree algorithm using flooding **Example**

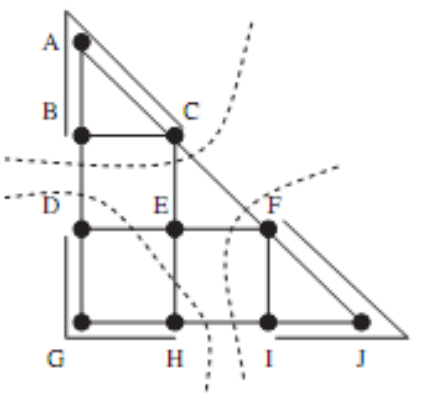


Figure 3 (source [3]) Asynchronous concurrent initiator example

Asynchronous concurrent-initiator spanning tree algorithm using flooding

- **Option 2**
- Suppress the instance initiated by one root and continue the instance initiated by the other root, based on some rule such as tie-breaking using the processor identifier
  - It must be ensured that the rule is correct

Asynchronous concurrent-initiator spanning tree algorithm using flooding

- **Example**
- A's initiation is suppressed due to the conflict detected along BD,
- G's initiation is suppressed due to the conflict detected along HI, and
- J's initiation is suppressed due to the conflict detected along CF,

  => the algorithm hangs and a tie rule must be used
- Algorithm 3 uses Design 2 option

Asynchronous concurrent-initiator spanning tree algorithm using flooding
Algorithm description

- Tie rule
  - Allows only the algorithm initiated by the root with the higher processor identifier to continue
- To implement this:
  - The messages need to be enhanced with a parameter that indicates the root node which initiated that instance of the algorithm

Asynchronous concurrent-initiator spanning tree algorithm using flooding
Algorithm description

```
(local variables)
int parent, myroot ⟵ ⊥
set of int Children, Unrelated ⟵ ∅
set of int Neighbors ⟵ set of neighbors
(message types)
QUERY, ACCEPT, REJECT
(1)   When the node wants to initiate the algorithm as a root:
(1a)  if (parent = ⊥) then
(1b)      send QUERY(i) to all neighbors;
(1c)      parent, myroot ⟵ i.

(2)   When QUERY(newroot) arrives from j:
(2a)  if myroot < newroot then   // discard earlier partial execution due
                                 // to its lower priority
(2b)      parent ⟵ j; myroot ⟵ newroot; Children, Unrelated ⟵ ∅;
(2c)      send QUERY(newroot) to all neighbors except j;
(2d)      if Neighbors = {j} then
(2e)          send ACCEPT(myroot) to j; terminate.   // leaf node
(2f)  else send REJECT(newroot) to j.
          // if newroot = myroot then parent is already identified.
          // if newroot < myroot ignore the QUERY. j will update its root
          // when it receives QUERY(myroot).

(3)   When ACCEPT(newroot) arrives from j:
(3a)  if newroot = myroot then
(3b)      Children ⟵ Children ∪ {j};
(3c)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3d)          if i = myroot then
(3e)              terminate.
(3f)          else send ACCEPT(myroot) to parent.
      // if newroot < myroot then ignore the message. newroot > myroot
      // will never occur.

(4)   When REJECT(newroot) arrives from j:
(4a)  if newroot = myroot then
(4b)      Unrelated ⟵ Unrelated ∪ {j};
(4c)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4d)          if i = myroot then
(4e)              terminate.
(4f)          else send ACCEPT(myroot) to parent.
      // if newroot < myroot then ignore the message. newroot > myroot
      // will never occur.
```

Algorithm 3 (source [3]) Spanning tree algorithm (asynchronous) without assuming a designated root. Initiators use flooding to start the algorithm. The code shown is for processor Pi where $1 \leq i \leq n$

Asynchronous concurrent-initiator spanning tree algorithm using flooding
Algorithm description

- When a **QUERY(newroot)** from **j** arrives at **i**, there are three possibilities:

1. **newroot > myroot**
   - Process **i** should suppress its current execution due to its lower priority
   - It reinitializes the data structures and joins **j**'s subtree with **newroot** as the root
2. **newroot = myroot**
   - **j**'s execution is initiated by the same **root** as **i**'s initiation, and **i** has already identified its parent.

   => A REJECT is sent to **j**
3. **newroot < myroot**
   - **j**'s **root** has a lower priority and hence **i** does not join **j**'s subtree
   - **i** sends a REJECT
   - **j** will eventually receive a QUERY(myroot) from **i** and abandon its current execution in favour of **i**'s **myroot** (or a larger value).

Asynchronous concurrent-initiator spanning tree algorithm using flooding
Algorithm description

- When an **ACCEPT(newroot)** from **j** arrives at **i**, there are three possibilities:

1. **newroot = myroot**
   - The ACCEPT is in response to a QUERY sent by i
   - The ACCEPT is processed normally
2. **newroot < myroot**
   - The ACCEPT is in response to a QUERY i had sent to j earlier, but i has updated its myroot to a higher value since then
   - Ignore the ACCEPT message.
3. **newroot > myroot**
   - The ACCEPT is in response to a QUERY i had sent earlier
   - But i never updates its myroot to a lower value.

   => This case cannot arise

- **Note**. The three possibilities when a REJECT(newroot) from j arrives at i are the same as for the ACCEPT message

Asynchronous concurrent-initiator spanning tree algorithm using flooding

- **Termination**
  - Main algorithm drawback
    - » only the root knows when its algorithm has terminated
  - To inform the other nodes, the root can send a special message along the newly constructed spanning tree edges
- **Complexity**
  - The time complexity of the algorithm is O(**l**) messages, and
  - The number of messages is O(**nl**)

4. Broadcast and converge-cast on a tree

- How spanning trees are used
  - distributing data/knowledge to nodes (via a broadcast)
  - collecting data/knowledge (via a converge-cast) from all nodes
- Figure 4 - generic graph with a spanning tree illustrating
  - the converge-cast type operations and
  - broadcast type operations

Broadcast and converge-cast on a tree

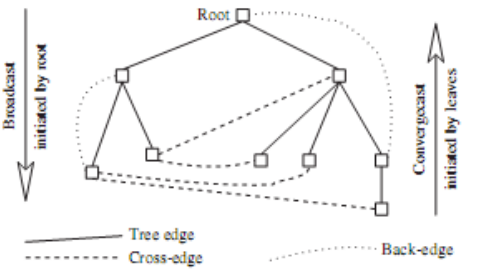


Figure 4 Broadcast and converge cast
in a spanning tree of a graph (source [3])

Broadcast and converge-cast on a tree
Broadcast algorithm

- A broadcast algorithm on a spanning tree - specified by rules BC1 and BC2
- **Rule BC1**
  - Tree root:
    - » sends the information to be broadcast to all its children
    - » Terminate.
- **Rule BC2**
  - When a (non-root) node receives information from its parent:
    - » Copy and then forward it to its children
    - » Terminate

Broadcast and converge-cast on a tree
Converge-cast algorithm

- Objective
  - collects information from all the nodes at the root node in order to compute some global function
- Algorithm initiation
  - Initiated by the leaf nodes of the tree
    - » usually in response to receiving a request sent by the root using a broadcast

Broadcast and converge-cast on a tree
Converge-cast algorithm

- The algorithm is specified by three rules CVC1, CVC2, CVC3
- **Rule CVC1 (Leaf node rule)**
  - Send its data/knowledge to its parent.
  - Terminate.
- **Rule CVC2 (non-leaf node and not root)**
  - When data/knowledge is received from all the child nodes:
    - » Integrate data/knowledge
    - » Sent to the parent
    - » Terminate
- **Rule CVC3 (root)**
  - When data/knowledge is received from all the child nodes
    - » Evaluate the global function
    - » Terminate

Broadcast and converge-cast on a tree

- Termination
  - The termination condition for each node in a broadcast as well as in a converge-cast is self-evident.
- Complexity
  - Space complexity
    - » Each broadcast and each converge-cast requires **n−l** messages
  - Time complexity
    - » Each broadcast and each converge-cast time is proportional to maximum height **h** of the tree which is **O(n)**

Broadcast and converge-cast on a tree
Examples

- Example 1 (converge-case)
  - Assumption
    - » Each node has an integer variable associated with the application
  - Objective
    - » Calculate the minimum of these variables
  - How
    - » Each leaf node reports its local value to its parent
    - » When a non-leaf node receives a report from all its children
      - ◆ Computes the minimum of those values
      - ◆ Sends this minimum value to its parent

Broadcast and converge-cast on a tree **Examples**

- Example 2 (converge-cast)
  - Objective
    - » Solving the leader election problem (details in a future lecture)
  - How
    - » Leader election - all the processes agree on a common distinguished process (the leader)
  - A leader is required in many distributed systems and algorithms because
    - » algorithms are typically not completely symmetrical
    - » some process has to take the lead in initiating the algorithm
    - » another reason – it is not desirable that all processes replicate the algorithm initiation, to save on resources

5. Single source shortest path algorithm: synchronous Bellman–Ford

- Bellman–Ford sequential shortest path algorithm
  - Finds the shortest path from a given node to all other nodes
  - Alternative solution to this problem: Dijkstra's algorithm
  - **Note.** Belman-Ford searches the shortest path by searching in ascending order of hops (different from Djikstra's)
  - Usage: DVR (Distance Vector Routing) Routing based on BF algorithm
- Network topology representation
  - A weighted graph, with unidirectional links
  - Weights (positive) may be lengths, delays or loads on the links

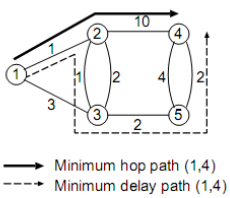


Fig. Source [Cavendish, Gerla]

Single source shortest path algorithm: synchronous Bellman–Ford

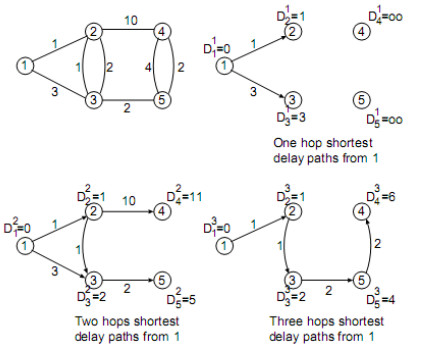


Fig 6 Belman-Ford algorithm execution hops (source Cavendish, Gerla)

Single source shortest path algorithm: synchronous Bellman–Ford

- Algorithm 4
  - A synchronous distributed algorithm to compute the shortest path
  - Assumptions
    - » The full topology (N, L) is not known to any process;
    - » No cyclic paths having negative weight
    - » Each process can communicate only with its neighbors
    - » Each process is aware of only the incident links and their weights
    - » The processes know the number of nodes | N | = n
      => The algorithm is not uniform (this assumption on n is required for termination)

Single source shortest path algorithm: synchronous Bellman–Ford

```
(local variables)
int length ⟵ ∞
int parent ⟵ ⊥
set of int Neighbors ⟵ set of neighbors
set of int {weight_{i,j}, weight_{j,i} | j ∈ Neighbors} ⟵ the known values of
   the weights of incident links
(message types)
UPDATE
(1)  if i = i_0 then length ⟵ 0;
(2)  for round = 1 to n − 1 do
(3)        send UPDATE(i, length) to all neighbors;
(4)        await UPDATE(j, length_j) from each j ∈ Neighbors;
(5)        for each j ∈ Neighbors do
(6)              if (length > (length_j + weight_{j,i}) then
(7)                    length ⟵ length_j + weight_{j,i}; parent ⟵ j.
```

Algorithm 4 – The single source synchronous distributed Bellman-Ford shortest path algorithm. The source is i0. The code is shown for processor Pi where $1 \leq i \leq n$ (source [3])

Single source shortest path algorithm: synchronous Bellman–Ford **Discussion**

- After the first round
  - The *length* variable of all nodes one hop away from the root in the final minimum spanning tree (MST) are calculated
- After k rounds
  - The *length* variable of all the nodes up to k hops away from the root in the final MST are calculated
  - Each node has its *length* variable set to the length of the shortest path consisting of at most k hops
  - The *parent* variable points to the parent node along such a path
    - » Parent field is used in the routing table to route to i_0 (the source node)

Single source shortest path algorithm: synchronous Bellman–Ford

- Termination
  - As the longest path can be of length **n−1**, the values of all variables are calculated after **n−1** rounds.
- Complexity
  - Time complexity is **n−1** rounds
  - Message complexity is **(n− 1) l** messages

Single source shortest path algorithm: synchronous Bellman–Ford

Case of dynamic network graph (1)

- The real case
- The shortest path is required for routing
- The classical Distance Vector Routing (DVR) based on the synchronous Bellman-Ford algorithm needs changes
- The outer for loop runs indefinitely
- Parent and length variables never stabilize (due to the dynamic nature of the system)

Single source shortest path algorithm: synchronous Bellman–Ford

Case of dynamic network graph (2)

- Variable length is replaced by array LENGTH[1..n]
  - LENGTH[k] denotes the length measured from node k as source node
  - LENGTH vector is included in each UPDATE message
  - Now, the k-th component of the LENGTH received from node m indicates the length of the shortest path from m to root k
- Variable parent is replaced by an array PARENT[1..n]
  - PARENT[k] denotes the next hop to which to route a packet destined for k
  - The array PARENT acts as the routing table

6. Single source shortest path algorithm: asynchronous Bellman–Ford

- Same assumptions as in synchronous Bellman-Ford
- No termination condition for nodes
  - Exercise - modify the algorithm so that each node knows when the shortest path to itself was determined

Single source shortest path algorithm: asynchronous Bellman–Ford

(local variables)
int $length \longleftarrow \infty$
set of int $Neighbors \longleftarrow$ set of neighbors
set of int $\{weight_{i,j}, weight_{j,i} \mid j \in Neighbors\} \longleftarrow$ the known values of the weights of incident links

(message types)
UPDATE

(1) **if** $i = i_0$ **then**
(1a) $length \longleftarrow 0$;
(1b) **send** UPDATE(i_0, 0) to all neighbors; **terminate**.

(2) When UPDATE(i_0, $length_j$) arrives from j:
(2a) **if** ($length > (length_j + weight_{j,i})$) **then**
(2b) $length \longleftarrow length_j + weight_{j,i}$; $parent \longleftarrow j$;
(2c) **send** UPDATE(i_0, $length$) to all neighbors;

Algorithm 5 – The single source asynchronous distributed Bellman-Ford shortest path algorithm. The source is i0. The code is shown for processor Pi where $1 \leq i \leq n$ (source [3])

Single source shortest path algorithm: asynchronous Bellman–Ford

- Complexity
  - Exponential complexity
  - Space complexity: **O(c^n)** number of messages
  - Time complexity: **O(c^n* d)**
    - » c is a constant
- Notes
  - If all links have equal weight the algorithm effectively computes the minimum-hop path
  - The minimum-hop routing tables to all destinations are calculated using **O(n^2 * l)** messages

7. Other shortest paths algorithms

- All sources shortest paths - distributed Floyd-Warshall
  - Centralized algorithm
  - Distributed algorithm (proposed by Toueg)
- Assumption
  - No negative weight cycles

Other shortest paths algorithms

- Centralized Floyd-Warshall algorithm
  - Data Structures:
  - Uses two **n x n** matrices LENGTH and VIA
    - » LENGTH[i,j] represents the shortest path from i to j
    - » LENGTH[i,j] is initialized to the initial known conditions
    - » LENGTH[i,j] =
      - $weigh_{i,j}$ if i and j are neighbors
      - 0 if i = j
      - Infinit, otherwise
    - » VIA[i,j] is the first hop on shortest path from i to j
    - » VIA[j,j] =
      - j if i and j are neighbors
      - 0 if i = j
      - Infinit, otherwise

8. Other shortest paths algorithms

- Distributed Floyd-Warshall algorithm (by Toueg)
- Data structures
  - At each node i are stored:
    - » Row i of LENGTH
    - » Row i of VIA
    - » These data structures are updated at node i
- Main challenges of the distributed algorithm
  - How to remotely access node i data from other nodes?
  - How to synchronize the execution at different nodes?

9. Challenges in designing distributed graph algorithms

- The graph can change
  - If either there are link or node failures, or worse still, partitions in the network
  - The graph can also change when new links and new nodes are added to the network.
- The case of mobile systems
  - The presented algorithms either fail or require a more complicated redesign if we assume that the graph topology changes dynamically
  - The graph (N, L) changes dynamically in the normal course of execution of a distributed execution
  - The challenge of mobile systems additionally needs to deal with the new communication model
    - » Each node is capable of transmitting data wirelessly, and all nodes within a certain radius can receive it.
    - » This is the unit-disk radius model
- The presented algorithms need to be redesigned to accommodate such changes