Distributed Algorithms & book by Donald

- Global state, global time, non-determinism.
- Application: Telecommunication, distributed information processing, scientific computing, real time process control.
- Topology, undirected graph.
  Three levels of typology: physical, logical, superimposed.

Classification and basic concepts

- Application execution and control algorithm execution (Protocol).
- Centralized algorithm and distributed algorithms.

23/10/12.

- ~~Semeto~~ Symmetric and Asymetric algorithms.
- Anonymous algorithms: An anonymous system is a system in which neither processes nor processors use their identifiers to make any execution decisions in the distributed algorithm. An anonymous algorithm is an algorithm which runs on an anonymous system ~~in which~~ and therefore does not use process or processor identifiers in the code. In these we have structural elegance and these are hard to design.
- Uniform algorithms: A uniform algo. does not use n, the no. of processes in the system as a parameter in it's code. These allow scalability transparency.
- Adaptive algorithm: Consider the context of a problem X. In a system with n' nodes, let k, $k \leq n$ be the no. of nodes participating in the context of X. If complexity of the system can be expressed in terms of k rather than n, the algorithm is adaptive.

- Deterministic v/s Non-deterministic execution
- Execution inhibition : blocking communicating primitives freeze the local execution. Non blocking, inhibition or freezing protocols. global inhibiting - send/ receive inibitory, internal event inhibitory.
- Synchronous / Asynchronous Systems :

  For synch. systems -

  i) known upper bound on the msg communication delay
  ii) known bounded drift rate for the clock.
  iii) known upper bound to execute a logical step in execution.

- Online v/s offline algorithms : based on data.
- Failure Models : t-fault tolerant

  Process failure models : Fail-stop, crash, receive omission, send ommision, general omission,

  Byzantine failure models or malicious faliures with auth,

  Byzantine or malicious failures

  Communication failure models : crash failure, omission failures, byzantine failures

  wait-free algorithms : resilient, fault-tolerant

- Communication channels : FIFO or non-FIFO.
- complexity measures and metrics :

  For sequential algorithms : time & space complexity, lower bound (Ω, ω), Upper bound (O, o)

and exact bound (Θ).

For distributed algorithms : time & space complexity, message complexity, space-complexity per node, system-wide space complexity, time-complexity per node, system-wide time complexity, Msg complexity (no. of msgs, ~~and~~ size of msgs and msg time complexity).

- Program Structure : Proposed by Hoare, CSP (Concurrent sequential processes)

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

26/10/19.

Synchronous single-initiator spanning tree algorithm using flooding. (Kshem)

Synchronous BFS spanning tree algorithm.

The code shown is for process P_i, $1 \leq i \leq n$.

(local variables).

int visited, depth ← 0.

int parent ← 1

Set of int Neighbors ← set of neighbors.

(message type)

QUERY.

if i = root then

visited ← 1,

depth ← 0

send QUERY to Neighbors;

for round = 1 to diameter do

if visited = 0 then

if any QUERY messages arrive then

parent ← randomly select a node from which QUERY was received.

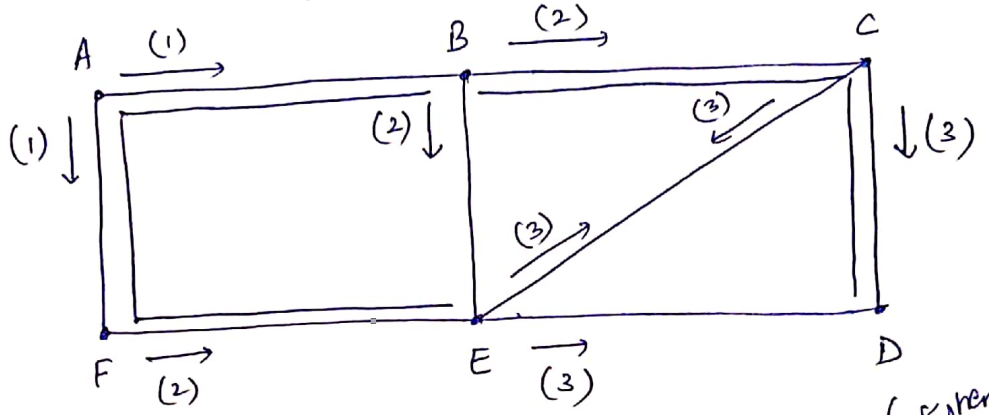
visited ← 1

depth ← round

send QUERY to Neighbors \ {senders of QUERYs received in this round}. ← excluding

delete any QUERY msgs that arrived in this round.

Example.



(Korem)

Asynchronous single-initiator spanning tree algorithm

The code for process P_i, $1 \le i \le n$

(local variables)

int parent ← ⊥ ← undefined

set of int children, unrelated ← ∅

set of int Neighbors ← set of neighbors

(Message types)

QUERY, ACCEPT, REJECT

When the predesignated root node wants to initiate the algo.

if (i = root and parent = ⊥) then

Send QUERY to all neighbors

~~par-do~~ parent = i

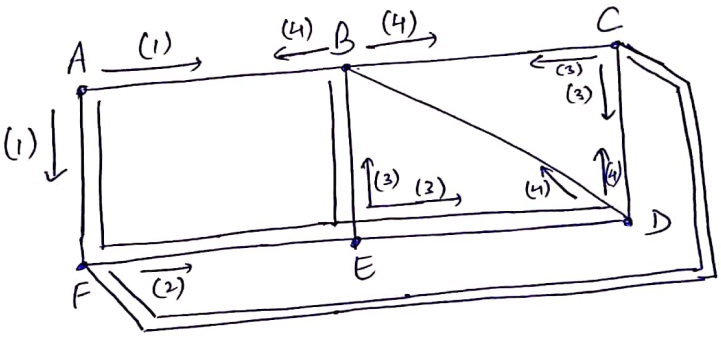
when QUERY arrives from j :

if parent = ⊥ then
    parent = j
    send ACCEPT to j;
    send QUERY to all neighbors except j;
    if (children ∪ unrelated) = (Neighbors \ {parent}) then terminate;

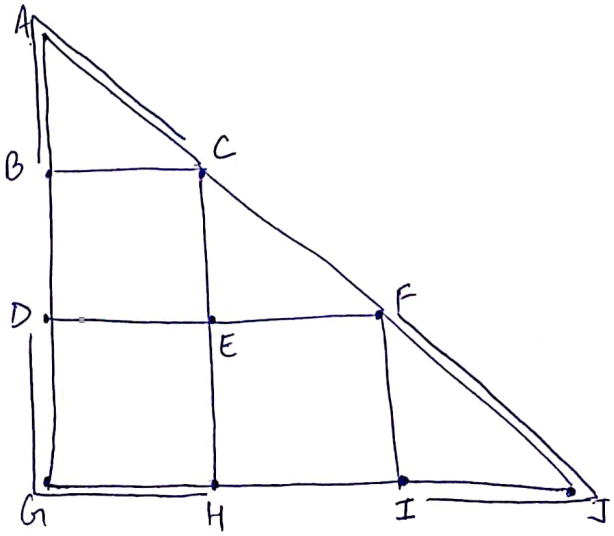
when ACCEPT arrives from j:
    children ← children ∪ {j};
    if (children ∪ unrelated) = (Neighbors \ {parent}) then terminate;

when REJECT arrives from j:
    unrelated ← unrelated ∪ {j};
    if (children ∪ unrelated) = (Neighbors \ {parent}) then terminate;

Example.



Asynchronous concurrent-initiator spanning tree algorithm using flooding. (Ishem)



(local variables)

int parent, myroot ← ⊥

set of int children, unrelated ← ∅

set of ~~neig~~ int Neighbors ← set of neighbors

*(message types)

QUERY, ACCEPT, REJECT

When the node wants to initiate the algorithm as a root:

if (parent = ⊥) then

send QUERY(i) to all neighbors;

parent, myroot ← i

when QUERY (newroot) arrives from j:

if (myroot < newroot) ~~then~~ ~~discard earlier partial execution~~

// discard earlier partial execution due to its low priority.

then parent ← j;

myroot ← newroot;

unrelated ← ∅;

send QUERY (newroot) to all neighbors except j;

if Neighbors = {j} then // leaf node.

send ACCEPT (myroot) to j;

terminate;

else send REJECT (newroot) to j;

// if newroot = myroot then parent is already identified

// if newroot < myroot then ignore the QUERY. j will update

// its root when it receives QUERY (myroot).

When ACCEPT (newroot) arrives from j :

if (newroot = myroot) then

children ← children ∪ {j}

if (children ∪ unrelated) = (Neighbors \ {parent}) then ~~terminate~~

if (i = myroot) then terminate

else Send ACCEPT (myroot) to parent.

// if newroot < myroot then ignore the msg and newroot > myroot

// will never occur.

When REJECT (newroot) arrives from j :

if (newroot = myroot) then

unrelated ← unrelated ∪ {j}.

if (children ∪ unrelated) = (Neighbors \ {parent}) then

if (i = myroot) then terminate

else Send ACCEPT (myroot) to parent.

// if newroot < myroot then ignore the msg and newroot > myroot

// myroot will never occur.

30/10/19.

Asynchronous concurrent-initiator DFS spanning tree algorithm

(local variables)

int parent, myroot ← ⊥;

set of int children, ~~unrelated~~ ← ∅;

set of int Neighbors, unkown ← set of neighbors

(message types)

QUERY, ACCEPT, REJECT

when the node wants to initiate the algorithm as root.

if (parent = ⊥) then send QUERY(i) to i (itself).

When QUERY(newroot) arrives from j:

if myroot < newroot then

parent ← j; myroot ← newroot, unknown ← set of neighbors;

unknown ← unkown \ {j};

if unknown ≠ 0 then

delete some x from unkown.

send QUERY(myroot) to x.

else send ACCEPT(myroot) to j;

else if (myroot = newroot) then

send REJECT(myroot) to j;

// if newroot < myroot ignore the query.

// j will update its root to higher root identifier when it receives

// its QUERY.

When ACCEPT(newroot) or REJECT(newroot) arrives from j:

if (newroot = myroot) then

if ACCEPT arrived then

children ← children ∪ {j}.

if (unkown = 0) then

if parent ≠ i then

send ACCEPT(myroot) to parent;

else set i as the root; terminate

else delete some x from unkown;

send QUERY(myroot) x;

// if newroot < myroot ignore the query. Since sending QUERY to
// j, i has updated its myroot. j will update its myroot to a
// higher root identifier. when it a query initiated by it
// newroot > myroot will never occur.

* Michael T. Goodrich and Roberto Tamassia "Algorithm design".

the Bellman-Ford Shortest path algorithm

It performs n-1 times a relaxation of every edge in the graph.

Input: A weighted directed graph $\vec{G}$ with n vertices and a vertex v of $\vec{G}$.

Output: A label D[u] for each vertex u of $\vec{G}$ such that D[u] is distance from v to u.

```
D[v] ← 0,
for each vertex u≠v of G do,
    D[u] = +∞.
for i ← 1 to n-1 do,
    { perform the relaxation operation on (u,z) }.
    if D[u] + w(u,z) < D[z] then
        D[z] ← D[u] + w(u,z);
if there are no edges left with potential relaxation operation then
    return the label D[u] of each vertex u.
else return "G contains a negative-weight cycle".
```

Single source shortest path algo. : Synchronous Bellman-Ford.

(weighted graph ; unidirectional links).

(local variables)

int length ← ∞

int parent ← ⊥

set of int Neighbors ← set of neighbors

set of int {$weight_{i,j}$, $weight_{j,i}$ | j ∈ Neighbor} ← the known values of the incident links.

(message types)

UPDATE.

if $i = i_0$ then length ← 0,

for round = 1 to n-1 do,

send UPDATE (i, length) to all neighbors ;

await UPDATE (j, $length_j$) from each j ∈ Neighbors,

for each j ∈ Neighbors do

if (length > $length_j$ + $weight_{j,i}$) then

length ← $length_j$ + $weight_{j,i}$;

parent = j.

The source is i_0, the code is for process P_i ($1 \leq i \leq n$).

06/11/19.

Single source shortest path algorithm : Asynchronous Bellman-ford.

(No negative weight cycles).

(local variables)

int length ← ∞

set of int Neighbors ← set of neighbors

set of int {weight$_{i,j}$, weight$_{j,i}$} ← the known values of the weights of incident lines.

(message types)

UPDATE

if $i = i_0$ then

length ← 0;

send UPDATE (i_0, 0) to all neighbors terminals.

when ~~UPATE~~ UPDATE (i_0, length$_j$) arrives from j.

if (length > (length$_j$ + weight$_{j,i}$)) then

length ← length$_j$ + weight$_{j,i}$; parent ← j

Send UPDATE (i_0, length) to all neighbors.

Distributed file Systems.

✓ A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network.

✓ The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks.

✓ Sharing of resources - key goal for distributed systems. Sharing of stored information - the most important aspect of distributed resource sharing.

✓ The design of large scale wide area read-write file storage systems poses problems of load balancing reliability, availability and security.

✓ Basic of distributed file system - emulate the functionality of non-distributed file system.

- Storage file system and their properties

	Sharing	Persistence	Distributed cache/ replicas	Consistency/ Maintenance	Example
Main Memory	X	X	X	1	RAM
File System	X	✓	X	1	Unix File System
Distributed File System	✓	✓	✓	✓	SUN NFS
Web	✓	✓	✓	X	Web Server
Distributed Shared memory	✓	X	✓	✓	
Remote objects (RMI/ORB)	✓	X	X	1	CORBA
Persistent object store	✓	✓	X	1	CORBA persistent state service
Peer-to-peer storage system	✓	✓	✓	2	

Types of consistency -

(1.: strict one copy) (✓: slightly weaker guarantee)

(2.: considerably weaker guarantee)

✓ Characteristics of the system file system.

→ responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface.

✓ File attribute record structure

File length
creation time stamp
read timestamp
write timestamp
attribute timestamp
reference count ↓ Managed by file system.

owner
file type
access control list ↓ user program.

✓ File system modules (for non-distributed file system).

Directory module : relates filenames to the IDs.
File modules : relates file IDs to particular files
Access control module : checks permission for operation requested.
File access module : read or write the file data or attributes.
Block module : accesses and allocates disk blocks.
Device modules : performs disk I/o and buffering.

✓ Distributed file system requirements

- Initially access transparency and location transparency
- Performance, scalability, concurrency control, fault tolerance and security.
- Transparency: The design must balance the flexibility and scalability that derive from transparency against software complexity and performance.

Access transparency. Client programs should be unaware of the distribution of files.

Location transparency. Client programs should see a uniform file name space.

Mobility transparency. Neither client programs nor system administration tables in client nodes need not be changed when files are moved.

Performance transparency. Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Scaling transparency. The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.