IN.5022 — Concurrent and Distributed Computing

Graph Algorithms: Distributed Shortest Path

Prof. P. Felber

pascal.felber@unine.ch

Agenda

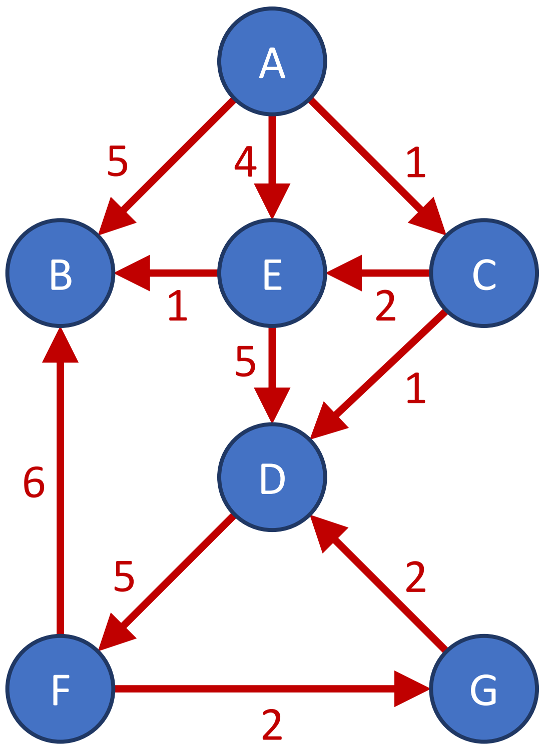
- Graph algorithms
- Distributed shortest path
  - Chandy-Misra’s algorithm
  - Support for negative weights/cycles

Graph algorithms

- Graph algorithms are at the core of distributed systems and networks: a "topology" can be represented by a graph
  - Internet routing relies on routers (vertices) interconnected via communication links (edges)
  - "Virtual" networks (e.g., social, Web) are also graphs
- Many problems can be solved using graph algorithms
  - Routing: shortest path computation
  - Broadcasting: spanning tree computation
  - Streaming: maximum flow computation
  - Robustness: disjoint paths computation
  - And more…

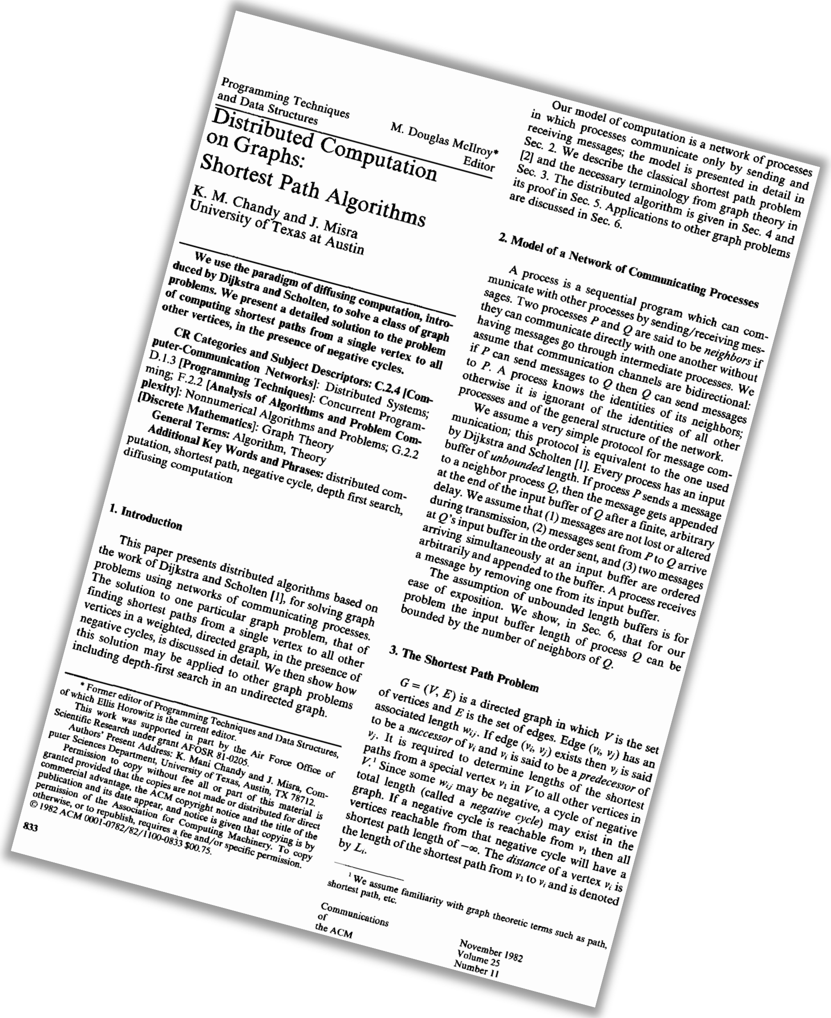
Shortest path in directed weighted graphs

- In a directed weighted graph G(V,E), what is the shortest distance L_i from a specific vertex v_1 to all other vertices v_i along the weighted edges of the graph?
  - Direct applicability to network routing, navigation systems, planning of itineraries…
  - Basic algorithm based on exploration [Dijkstra, 1959]
  - Algorithm for distributed shortest path computation [Chandy-Misra, 1982]



Example: Chandy-Misra's shortest path [1982]

- Distributed algorithm for shortest path
  - Refinement of the Bellman-Ford algorithm to compute ARPANET routes in the 60s-70s
  - Assumes a static network topology of nodes with asynchronous message passing
  - Each vertex v_j is an independent process p_j
  - Edges have weights (cost of communication)
  - Processes exchange message along edges
  - Process p_1 (vertex v_1) initiates computation
  - Each process $p_{j>1}$ can compute the shortest path (minimal cost) to the source node p_1



Programming Techniques and Data Structures

M. Douglas McIlroy*
Editor

Distributed Computation on Graphs: Shortest Path Algorithms

K. M. Chandy and J. Misra
University of Texas at Austin

We use the paradigm of diffusing computation, introduced by Dijkstra and Scholten, to solve a class of graph problems. We present a detailed solution to the problem of computing shortest paths from a single vertex to all other vertices, in the presence of negative cycles.

CR Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms: Algorithm, Theory

Additional Key Words and Phrases: distributed computation, shortest path, negative cycle, depth first search, diffusing computation

1. Introduction

This paper presents distributed algorithms based on the work of Dijkstra and Scholten [1], for solving graph problems using networks of communicating processes. The solution to one particular graph problem, that of finding shortest paths from a single vertex to all other vertices in a weighted, directed graph, in the presence of negative cycles, is discussed in detail. We then show how this solution may be applied to other graph problems including depth-first search in an undirected graph.

Our model of computation is a network of processes in which processes communicate only by sending and receiving messages; the model is presented in detail in Sec. 2. We describe the classical shortest path problem [2] and the necessary terminology from graph theory in Sec. 3. The distributed algorithm is given in Sec. 4 and its proof in Sec. 5. Applications to other graph problems are discussed in Sec. 6.

2. Model of a Network of Communicating Processes

A process is a sequential program which can communicate with other processes by sending/receiving messages. Two processes P and Q are said to be *neighbors* if they can communicate directly with one another without having messages go through intermediate processes. We assume that communication channels are bidirectional: if P can send messages to Q then Q can send messages to P. A process knows the identities of its neighbors; otherwise it is ignorant of the identities of all other processes and of the general structure of the network.

We assume a very simple protocol for message communication; this protocol is equivalent to the one used by Dijkstra and Scholten [1]. Every process has an input buffer of *unbounded* length. If process P sends a message to a neighbor process Q, then the message gets appended at the end of the input buffer of Q after a finite, arbitrary delay. We assume that (1) messages are not lost or altered during transmission, (2) messages sent from P to Q arrive at Q's input buffer in the order sent, and (3) two messages arriving simultaneously at an input buffer are ordered arbitrarily and appended to the buffer. A process receives a message by removing one from its input buffer.

The assumption of unbounded length buffers is for ease of exposition. We show, in Sec. 6, that for our problem the input buffer length of process Q can be bounded by the number of neighbors of Q.

3. The Shortest Path Problem

$G = (V, E)$ is a directed graph in which V is the set of vertices and E is the set of edges. Edge (v_i, v_j) has an associated length w_{ij}. If edge (v_i, v_j) exists then v_j is said to be a *successor* of v_i and v_i is said to be a *predecessor* of v_j. It is required to determine lengths of the shortest paths from a special vertex v_1 in V to all other vertices in V.[1] Since some w_{ij} may be negative, a cycle of negative total length (called a *negative cycle*) may exist in the graph. If a negative cycle is reachable from v_1 then all vertices reachable from that negative cycle will have a shortest path length of $-\infty$. The *distance* of a vertex v_i is the length of the shortest path from v_1 to v_i and is denoted by L_i.

* Former editor of Programming Techniques and Data Structures, of which Ellis Horowitz is the current editor.

This work was supported in part by the Air Force Office of Scientific Research under grant AFOSR 81-0205.

Authors' Present Address: K. Mani Chandy and J. Misra, Computer Sciences Department, University of Texas, Austin, TX 78712.

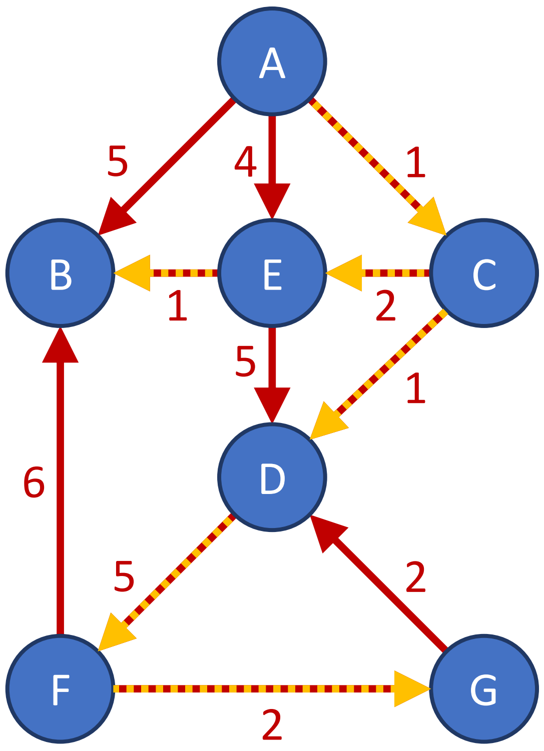
[1] We assume familiarity with graph theoretic terms such as path, shortest path, etc.

833
Communications of the ACM
November 1982 Volume 25 Number 11

https://citeseerx.ist.psu.edu/…
…viewdoc/summary?doi=10.1.1.104.8179

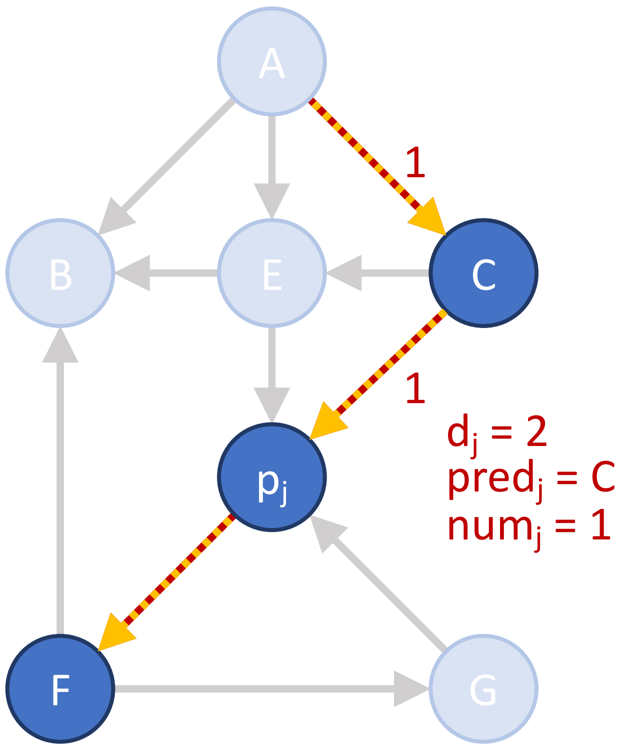
Algorithm: model

- Consider a graph $G=(V,E)$ representing a network of processes
  - Processes p_j and p_k are neighbours if the edge (v_j,v_k) or (v_k,v_j) exists in G
  - Process p_j only knows its successor neighbours N_j and the weight w_{jk} for each outgoing edge (v_j,v_k)
  - The best knowledge of node j about its distance to node 1 via the shortest path is denoted by d_j
  - We initially assume no negative edges



Algorithm: state

- Each process p_j maintains local state
  - d_j — initially = ∞
    Best known distance from p_1 computed so far by p_j at this point
  - $pred_j$ — initially = ⊥ (i.e., undefined)
    Predecessor node from which d_j was received, i.e., next to last vertex on the shortest path from p_1 to p_j computed so far
  - num_j — initially = 0
    Number of non-acknowledged messages sent by this process (ACKs needed to ensure termination and handle "negative cycles")



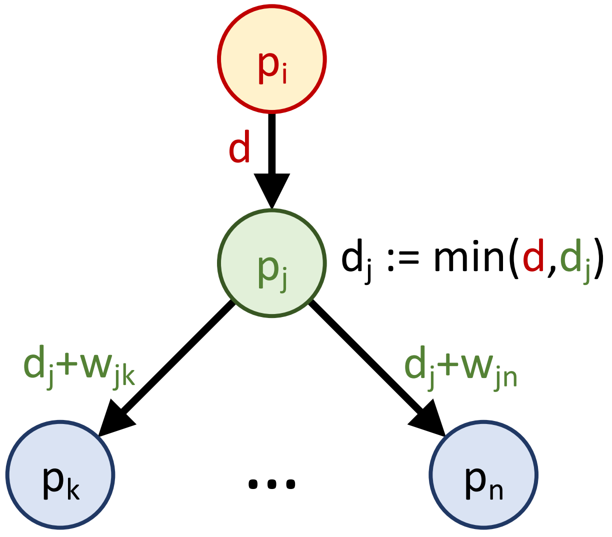
Algorithm: basic principle

- Processes incrementally compute their distance from the source p_1
- When $p_{j>1}$ receives distance d from p_i
  - If d is smaller than the best known distance d_j computed so far …

  …then update best known distance d_j…

  …remember parent $pred_j = p_i$ along path…

  …and send new distance $d_k = d_j + w_{jk}$ to each successor neighbour p_k
  - Otherwise do nothing
- Initially: p_1 sends w_{1k} to each $p_k \in N_1$



Algorithm: adding ACKs (termination)

- At source p_1
  - Set num_1 to $|N_1|$ after sending initial message to all successors
  - Upon receiving ACK, decrement num_1 and, if 0, terminate protocol (phase II) by propagating STOP message
- At non-source p_j
  - Increment num_j after sending new distance to each successor p_k
  - Upon receiving new distance from p_i, if smaller than d_j send ACK to previous $pred_j$ before updating it, otherwise send ACK to p_i
  - Upon receiving ACK, decrement num_j and, if 0, send ACK to $pred_j$
- Each distance message gets one ACK (immediate or delayed)

Pseudo-code: state and initialisation

- Distance d decreases from ∞ until converging to the shortest path
- Initially parent node pred on shortest path is undefined and there are no missing ACKs yet (num = 0)
- Variable s is the new (possibly shorter) distance received via incoming edge

```
program shortest path

define
  d, s : distance % s: new distance received
  pred : process
  num : integer
  N : set of processes          % N: successors

initially
  d = ∞
  pred = ⊥
  num = 0
```

Pseudo-code: source p_1

- The source process p_1 initiates the protocol
- Upon receiving new distance, simply send ACK (source cannot get better distance with non-negative weights)
- Source keeps track of missing ACKs and, when reaching 0, initiates phase II (propagate STOP message)

```
% Program for process p₁ (source)
∀pₖ ∈ N : send {w₁ₖ,1} to pₖ
num := |N|
do
» {s,i} → send ACK to pᵢ        % p₁ just acks
» ACK → num := num - 1  % one less ack to go
» num = 0 → ∀pₖ ∈ N : send STOP to pₖ, exit
od
```

Pseudo-code: regular nodes $p_{j>1}$

- Non-source process p_j gets new distance s from p_i
  - If better, we switch parent
    - Send ACK to previous parent
    - Update pred and d
    - Propagate new distances and update missing ACKs as needed
  - If worse, just send ACK
- Upon ACK, decrement num
- If no missing ACK, send ACK once to parent (termination)

```
% Program for process pj ≠ p1
do
» {s,i} AND s < d →   % new s from pi: better
  if num > 0 →
    send ACK to pred  % ack before switching
  fi
  pred := i             % switch to new parent
  d := s
  ∀pk ∈ N : send {d+wjk,j} to pk
  num := num + |N|
» {s,i} AND s ≥ d → send ACK to pi   % worse
» ACK → num := num - 1
» num = 0 AND pred ≠ ⊥ → send ACK to pred
» STOP → ∀pk ∈ N : send STOP to pk, exit
```

Complete pseudo-code of algorithm

```
program shortest path
define
  d, s : distance % s: new distance received
  pred : process
  num : integer
  N : set of processes        % N: successors
initially
  d = ∞, pred = ⊥, num = 0

% Program for process p1 (source)
∀pk ∈ N : send {w1k,1} to pk
num := |N|
do
» {s,i} → send ACK to pi          % p1 just acks
» ACK → num := num – 1  % one less ack to go
» num = 0 → ∀pk ∈ N : send STOP to pk, exit
od
```

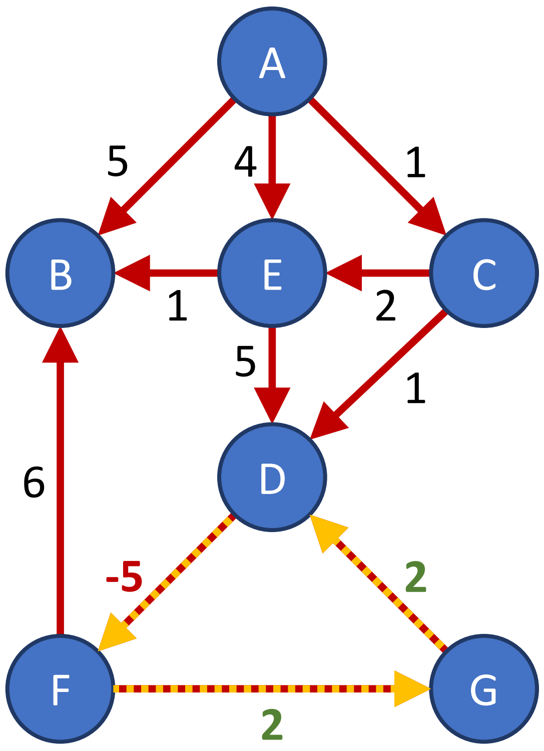
```
% Program for process pj ≠ p1
do
» {s,i} AND s < d →   % new s from pi: better
  if num > 0 →
    send ACK to pred   % ack before switching
  fi
  pred := i             % switch to new parent
  d := s
  ∀pk ∈ N : send {d+wjk,j} to pk
  num := num + |N|
» {s,i} AND s ≥ d → send ACK to pi    % worse
» ACK → num := num - 1
» num = 0 AND pred ≠ ⊥ → send ACK to pred
» STOP → ∀pk ∈ N : send STOP to pk, exit
```

Why does it work (without negative weights)?

- Intuitively… *[correctness]*
  - The best-known distance d at each process p_j only decreases
  - Whenever a node learns about a better distance, it informs its neighbours, hence providing them with possibly shorter paths
  - A node will change parent only if it receives a shorter path
  - When a node stops receiving new paths, d is the shortest distance from p_1 to p_j and its parent is the next to last node on that path
- Intuitively… *[termination]*
  - A node sends ACK when it has no outstanding ACKs downstream or when discarding a parent (if path is worse or before switching)
  - Hence each distance message sent will result in exactly one ACK

Supporting negative weights

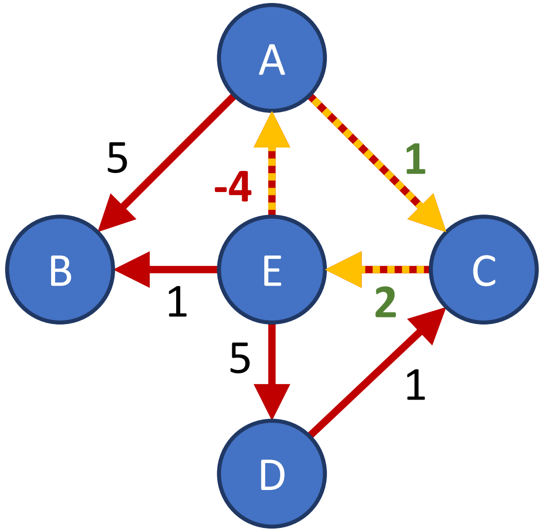
- Negative weights can create negative cycles that decrease the distance
  - The shortest path will become negative and ultimately tend toward $-\infty$
  - The challenge is to detect such cycles
- The Chandy-Misra algorithm exploits properties of ACKs (and distances)
  - In cycles, shorter distances are continuously sent without being acknowledged (num > 0)
  - Outside the cycle, all processes (incl. p_1) eventually have num = 0

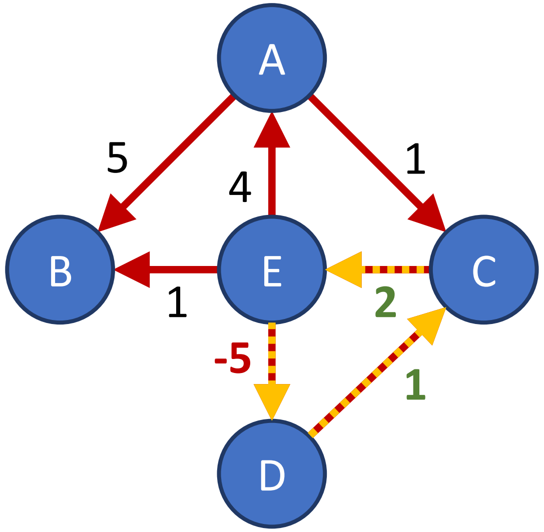


Supporting negative weights

- Principle: a cycle is detected by the source in 2 cases
  1. The source p_1 receives a negative path: p_1 is in a negative cycle ($num_1 > 0$)
  2. If the source initiates phase II with all ACKs received ($num_1 = 0$) and some node p_i has missing ACKs: p_i is in a negative cycle but not p_1

  The case of p_i receiving a negative path is subsumed by case 2
- Negative weights are not a problem if they do not create negative cycles (why?)



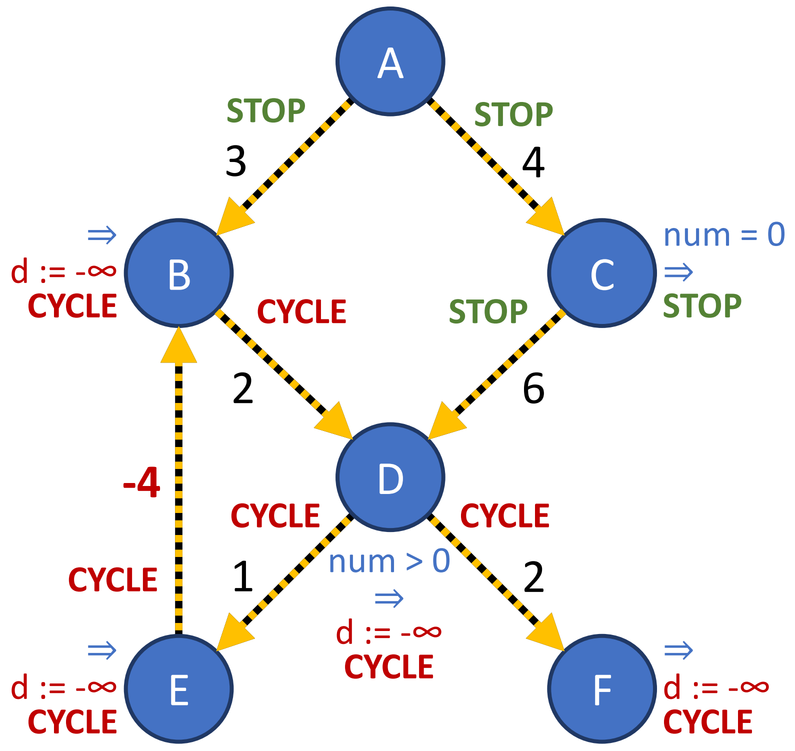


Adapting the algorithm for negative cycles

- Phase II propagates 2 types of messages
  - STOP ("Over?" in [CM82]): source starts phase II, nodes must check for negative cycles
  - CYCLE ("Over–" in [CM82]): negative cycle was detected
    - Nodes set d to $-\infty$ to indicate that cycle was found (avoid re-propagation)
- Source initiates phase II with STOP or CYCLE (if detected)
- When receiving STOP (node $p_{j>1}$)
  - If num = 0 and d $\neq -\infty$, propagate STOP
  - If num > 0 and d $\neq -\infty$, set d = $-\infty$ and propagate CYCLE
- When receiving CYCLE (node $p_{j>1}$)
  - If d $\neq -\infty$, set d = $-\infty$ and propagate CYCLE

Example

- Source starts phase II with regular STOP message
  - Normal propagation of STOP until encountering the cycle (B,D,E) at D
  - Node D detects cycles and switch to propagation of CYCLE message
  - Nodes only propagate CYCLE messages if they did not already (i.e., if d is not $-\infty$)
- If source detects cycle in phase I (by receiving a negative shorter path) it immediately propagates CYCLE in phase II



Summary

- Graph algorithms are ubiquitous in distributed systems
  - Application to all types of network problems (not just “physical”)
  - Classical model: vertices are processes, edges are channels
- A fundamental problem: distributed shortest path
  - Chandy-Misra algorithm [1982]
    - Simple version without termination (Bellman-Ford)
    - Adding termination
    - Handling negative cycles
  - Implementation in Erlang/Elixir is a direct mapping of pseudo-code