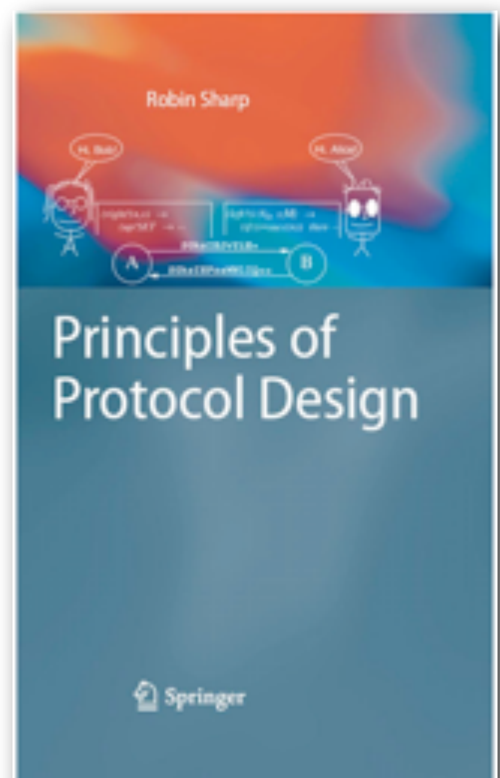
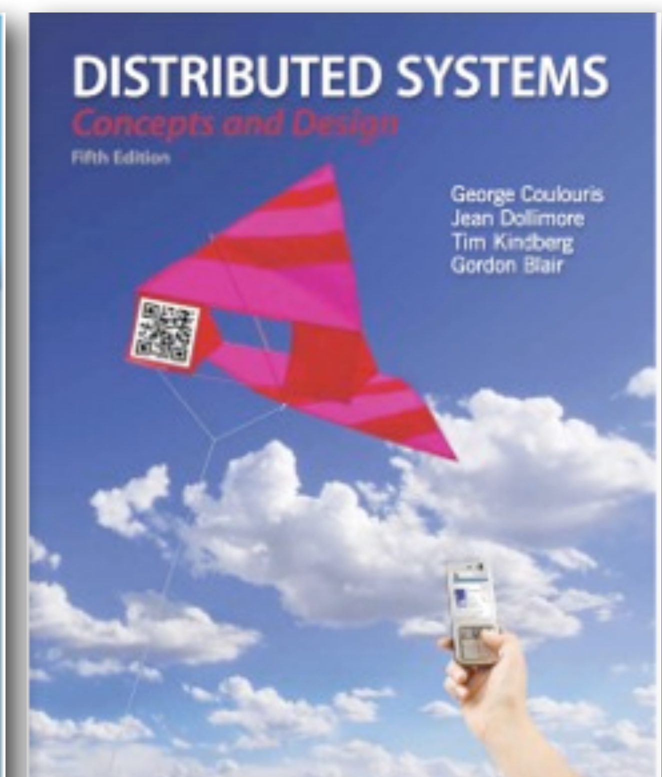
Global States

Nicola Dragoni
Embedded Systems Engineering
DTU Informatics

Introduction
Clock, Events and Process States
Logical Time and Logical Clocks
Global States





Outline

- Global State - *what is a global state of a distributed system?*
  - definition
  - next global state
- Distributed Snapshot - *how to record a global state of a distributed system?*
  - consistent global states
  - Chandy and Lamport's algorithm
- Evaluating Predicates - *why/how to use the recorded global states?*
  - evaluating Stable Predicates
  - evaluating Non Stable Predicates

Problem: Finding the Global State

- Problem: to find the global state of a distributed system (in which data items can move from one part of the system to another)

- Why? There are innumerable uses for this, for instance:
  - finding the *total number of files* in a distributed file system, where files may be moved from one file server to another
  - finding the *total space occupied by files* in such a distributed file system
  - in general, *to detect global properties of the distributed system*, such as garbage collection, deadlock, termination

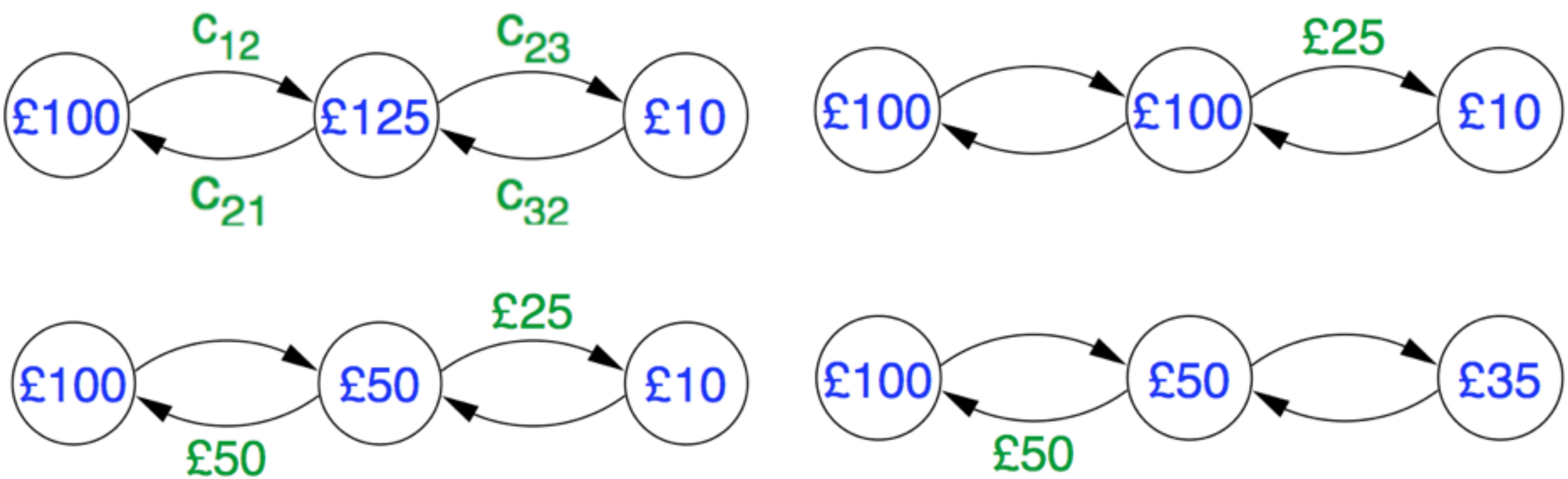
- Solution: distributed snapshot algorithm
  (Chandy and Lamport, 1985)

Global State

- Idea: global states are described by

  1) the states of the participating PROCESSES, together with

  2) the states of the CHANNELS through which data (i.e., the files) pass when being transferred between these processes



N.B.: $\sum$ Money = £235

Events

- Each event is described by 5 components: *e = <p, s, s', M, c>*
  - Process *p* goes from state *s* to state *s'*
  - Message *M* is sent or received on channel *c*
- Event *e = <p, s, s', M, c>* is only possible in global state *S* if:
  1. *p*'s state in *S* is just exactly *s*
  2. If *c* is directed towards *p*, then *c*'s state in *S* must be a sequence of messages with *M* at its head
- A *possible computation* of the system is a *sequence of possible events*, starting from the *initial global state* of the system

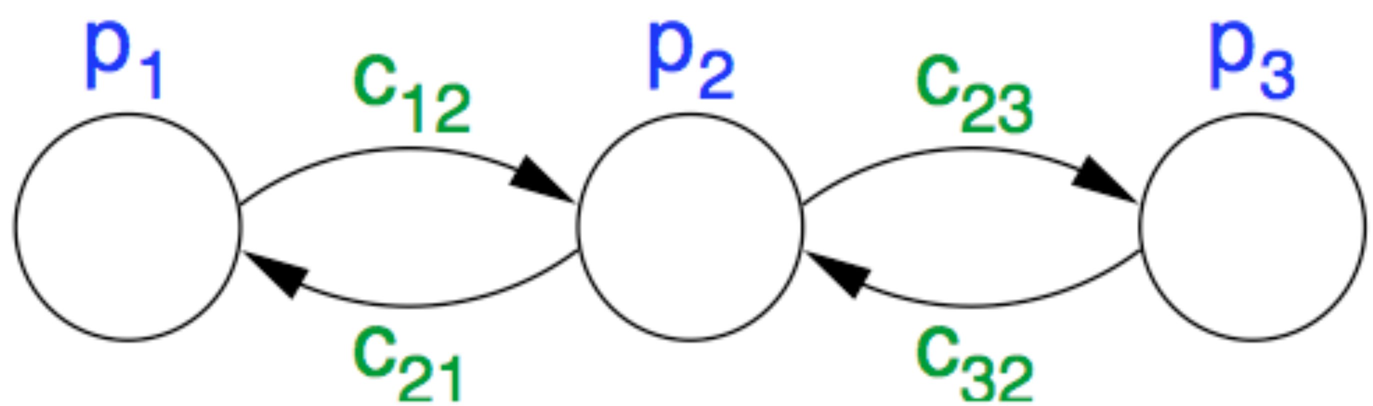
Next Global State

- If $e = \langle p, s, s', M, c\rangle$ takes place in global state S, then the following **global state** is $next(S, e)$, where:

1. p's state in $next(S, e)$ is s'
2. If c is directed towards p, then c's state in $next(S, e)$ is c's state in S, with M removed from the head of the message sequence
3. If c is directed away from p, then c's state in $next(S, e)$ is c's state in S, with M added to the tail of the message sequence

Example: A Possible Computation

- c_{ij} denotes the channel which can carry messages from p_i to p_j
- System configuration:

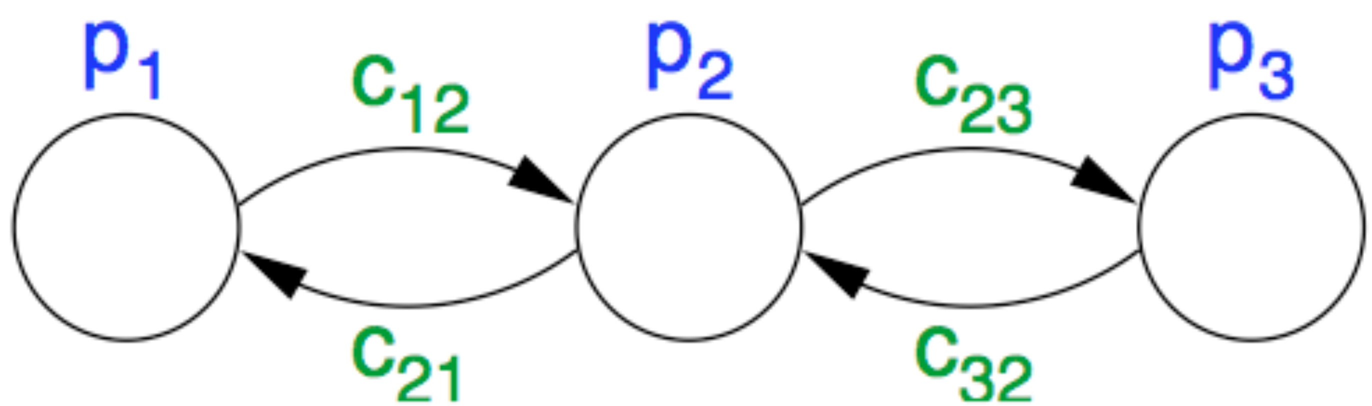


	Event					⇒	Global state S after event						
	P	s	s'	M	c		P_1	P_2	P_3	c_{12}	c_{21}	c_{23}	c_{32}
							100	125	10	⟨⟩	⟨⟩	⟨⟩	⟨⟩
e_1	P_1	100	25	75	c_{12}	⇒	25	125	10	⟨75⟩	⟨⟩	⟨⟩	⟨⟩
e_2	P_2	125	100	25	c_{23}	⇒	25	100	10	⟨75⟩	⟨⟩	⟨25⟩	⟨⟩
e_3	P_2	100	175	75	c_{12}	⇒	25	175	10	⟨⟩	⟨⟩	⟨25⟩	⟨⟩
e_4	P_2	175	125	50	c_{21}	⇒	25	125	10	⟨⟩	⟨50⟩	⟨25⟩	⟨⟩
e_5	P_3	10	35	25	c_{23}	⇒	25	125	35	⟨⟩	⟨50⟩	⟨⟩	⟨⟩
e_6	P_1	25	75	50	c_{21}	⇒	75	125	35	⟨⟩	⟨⟩	⟨⟩	⟨⟩

Outline

- Global State - *what is a global state of a distributed system?*
  - definition
  - next global state
- Distributed Snapshot - *how to record a global state of a distributed system?*
  - consistent global states
  - Chandy and Lamport's algorithm
- Evaluating Predicates - *why/how to use the recorded global states?*
  - evaluating Stable Predicates
  - evaluating Non Stable Predicates

[Distributed Snapshots] The Question



Can we now find rules for
when to take snapshots of
the individual processes and channels
so as to build up a *consistent* picture of
the global state *S*?

Assumptions

- The algorithm relies on two main assumptions:
  - Channels are ERROR-FREE and SEQUENCE PRESERVING (FIFO)
  - Channels deliver transmitted msgs after UNKNOWN BUT FINITE DELAY
- Other assumptions:
  - The only events in the system which can give rise to changes in the state are communicating events

[Distributed Snapshots] Consistent Picture

- Let us consider the happened-before relation
- If $e_1 \rightarrow e_2$ then e_1 happened before e_2 and could have caused it
- A consistent picture of the global state is obtained if we include in our computation a set of possible events, H, such that

$$e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$$

- If e_i were in H, but e_j were not, then the set of events would include the effect of an event (for instance, the receipt of a file), but not the event causing it (the sending of the file), and an inconsistent picture would arise

[Distributed Snapshots] Consistent Global State

- A consistent picture of the global state is obtained if we include in our computation a set of possible events, H, such that

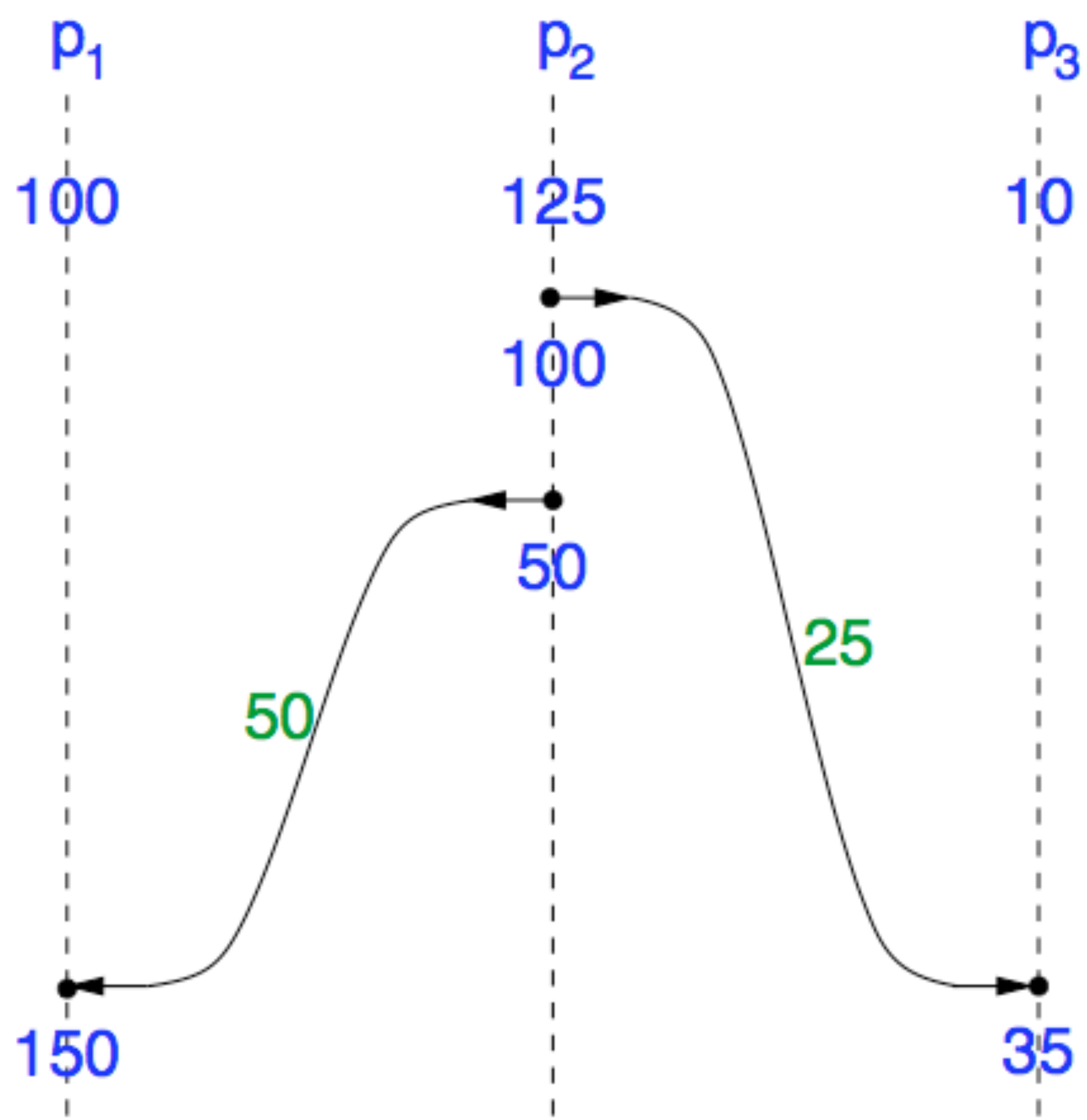
$$e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$$

- The consistent GLOBAL STATE is then defined by

  $GS(H)$ = The state of each process p_i after p_i's last event in H
  + for each channel, the sequence of msgs sent in H but not received in H

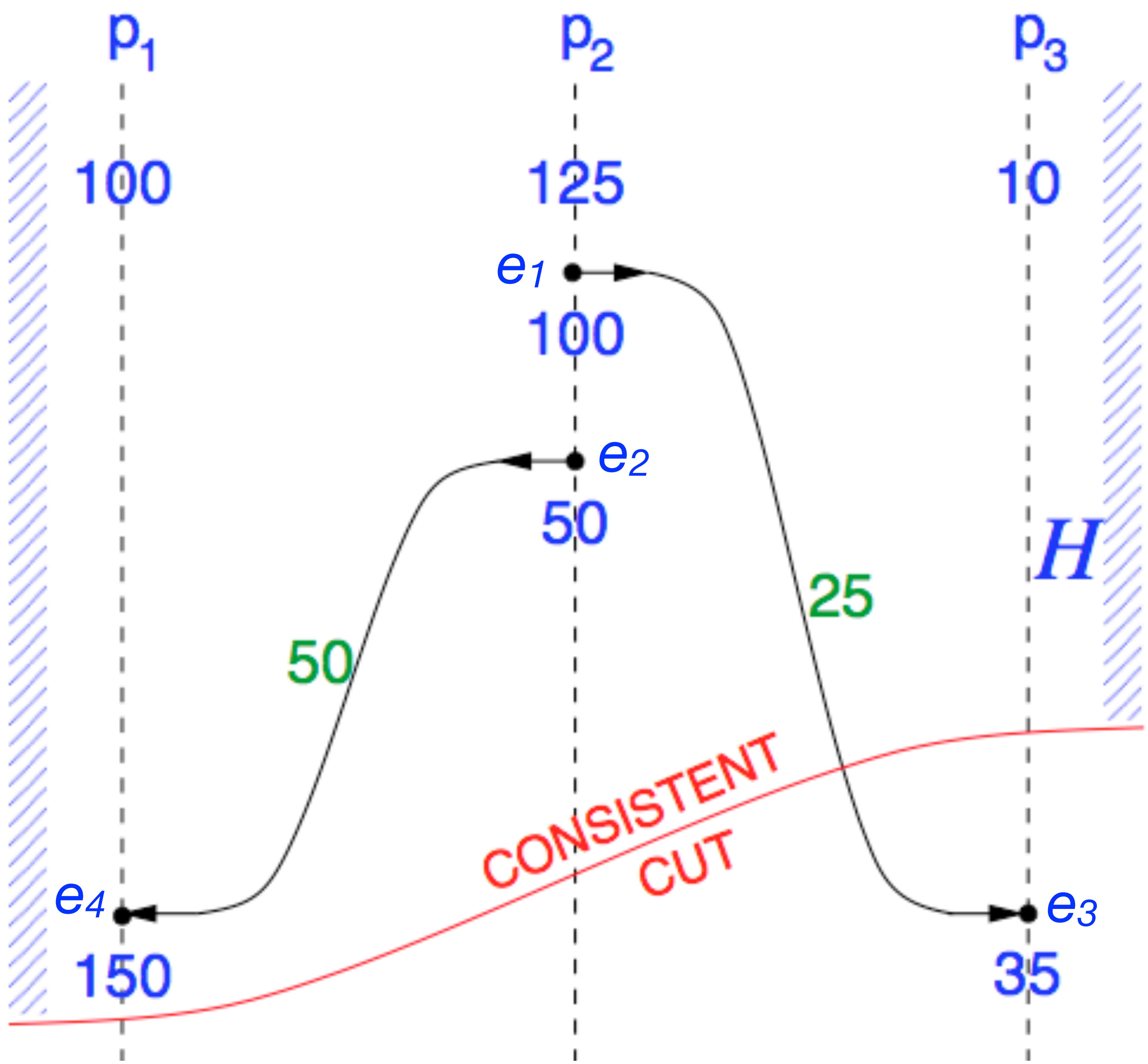
- In the distributed systems jargon, we say that *consistent global states are delimited by a "CUT" representing a consistent picture of the global state of the system*

Example: A Possible Computation



Example: Consistent Cut

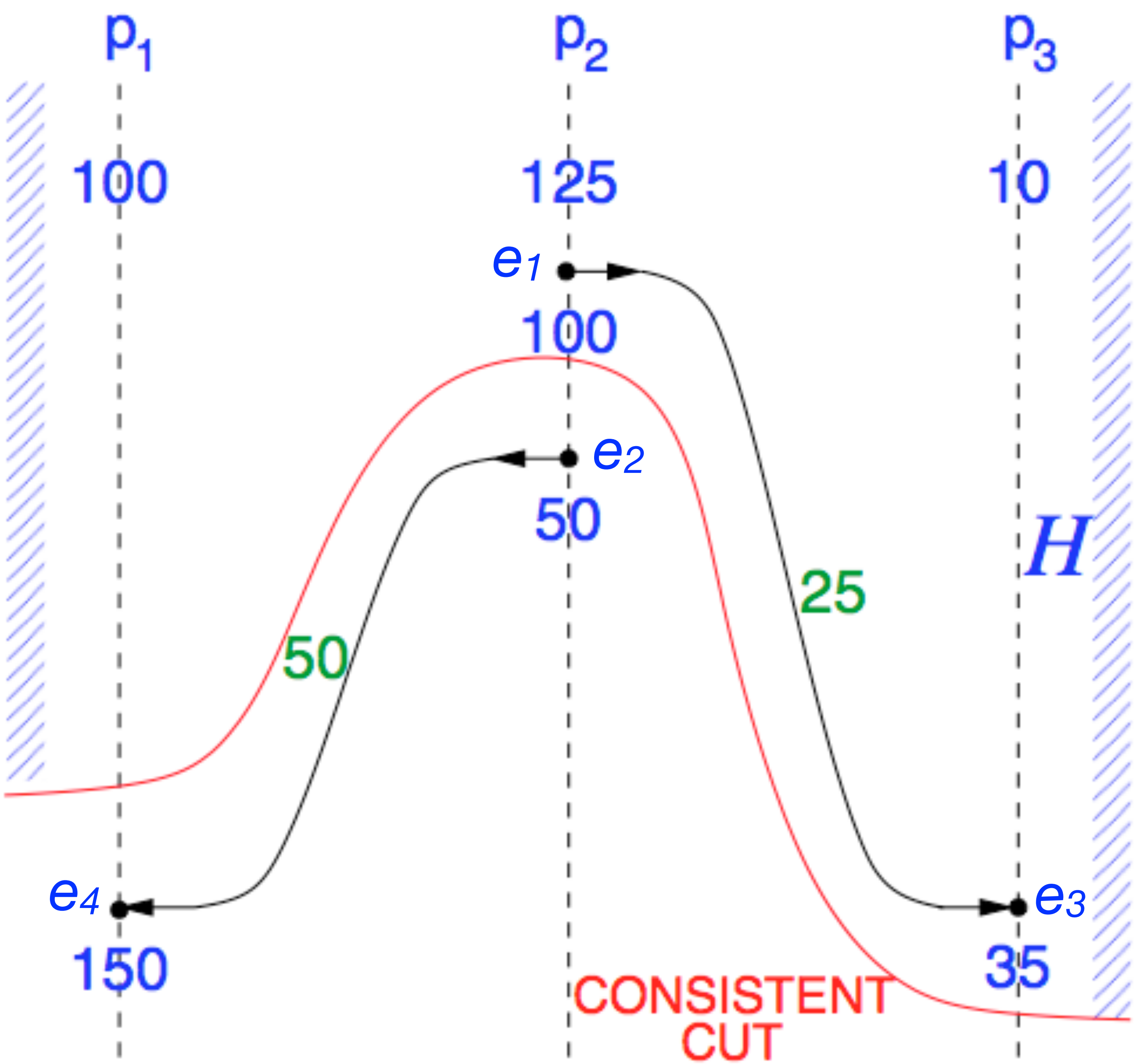
- REMEMBER: The CUT limiting H is defined by: $e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$



H contains $\{e_1, e_2, e_4\}$

Example: Consistent Cut

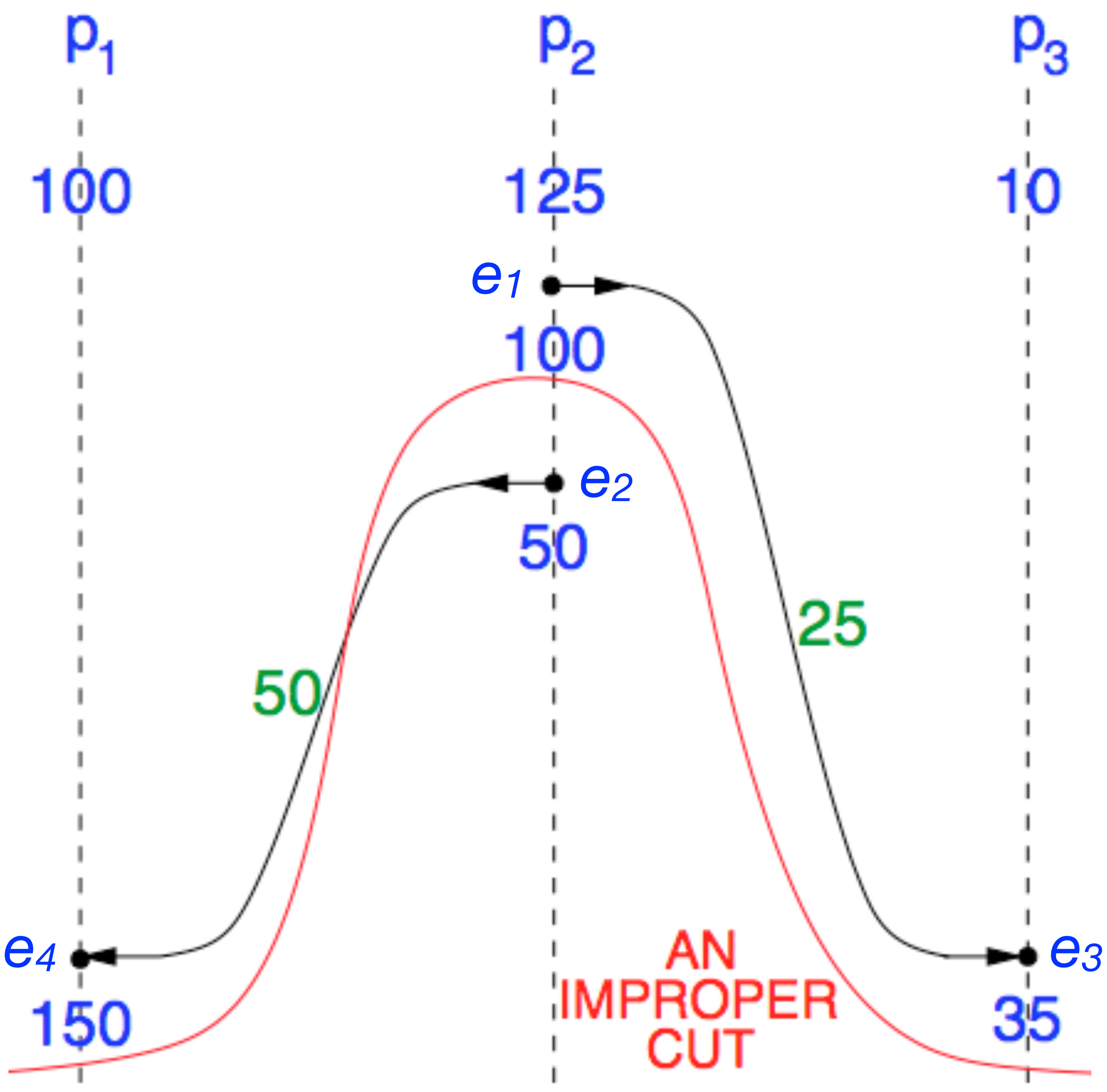
- REMEMBER: The CUT limiting H is defined by: $e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$



H contains $\{e_1, e_3\}$

Example: Inconsistent Cut

- REMEMBER: The CUT limiting H is defined by: $e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$



H contains $\{e_1, e_3, e_4\}$, but not e_2, where $e_2 \rightarrow e_4$

How to Construct *H*?

- Idea: The CUT and associated (consistent) set of events, *H*, are constructed by including specific control messages (MARKERS) in the stream of ordinary messages

- Remember that we assume that:
  - Channels are all FIFO channels
  - A transmitted marker will be received (and dealt with) within a FINITE TIME

Chandy and Lamport's Algorithm to Construct *H*

- Process p_i follows two rules

1. SEND MARKERS

   Record p_i's state
   Before sending any more messages from p_i, send a marker on each channel c_{ij} directed away from p_i

2. RECEIVE MARKER

   On arrival of a marker via channel c_{ji}:

   IF p_i has not recorded its state

   THEN SEND MARKERS rule; record c_{ji}'s state as empty

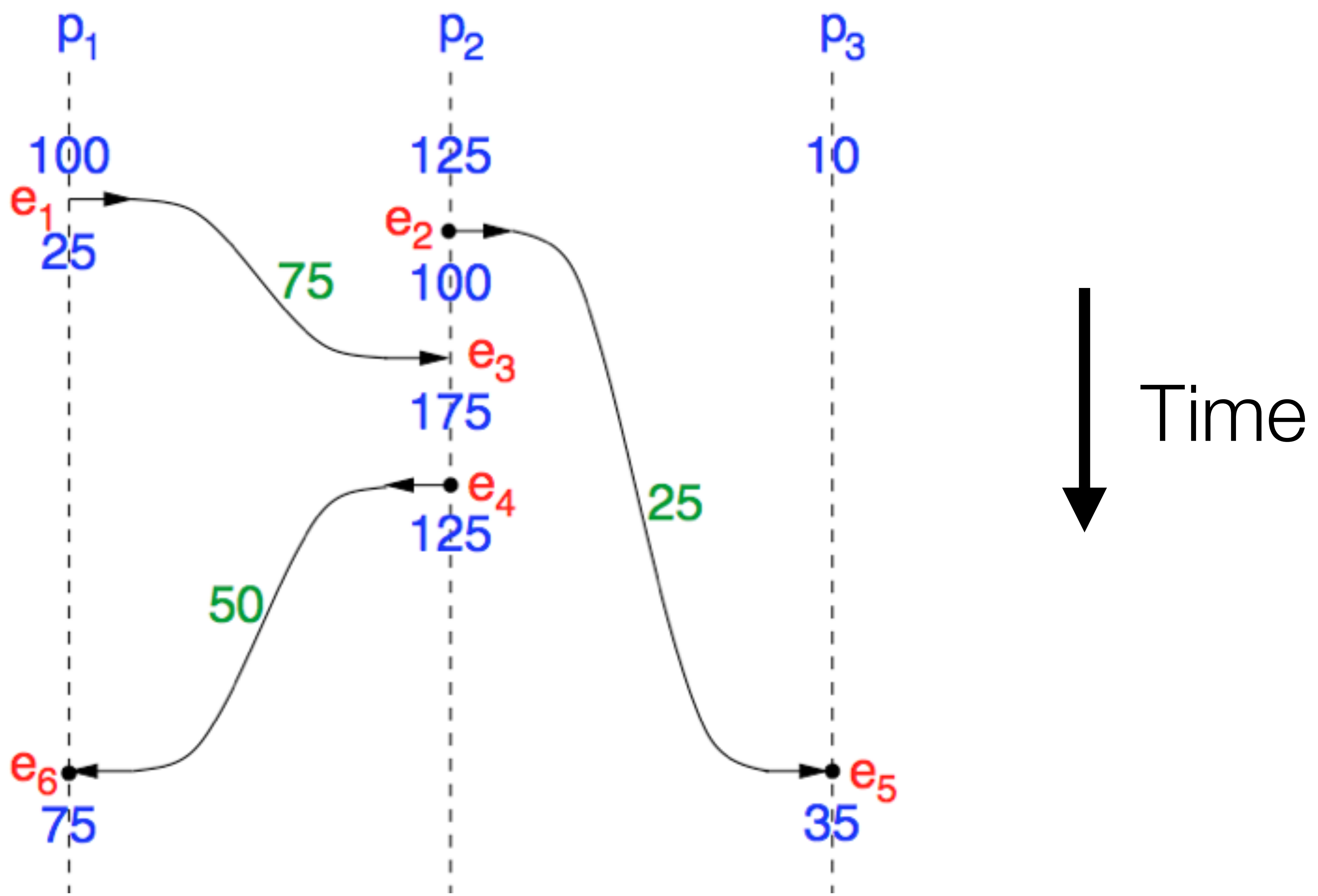
   ELSE record c_{ji}'s state as the sequence of messages received on c_{ji} since p_i last noted its state

Chandy and Lamport's Algorithm to Construct *H*

- The algorithm can be initiated by any process by executing the rule SEND MARKERS
  - Multiple processes can initiate the algorithm concurrently!
  - Each initiation needs to be distinguished by using unique markers
  - Different initiations by a process are identified by a sequence number
- The algorithm terminates after each process has received a marker on all of its incoming channels

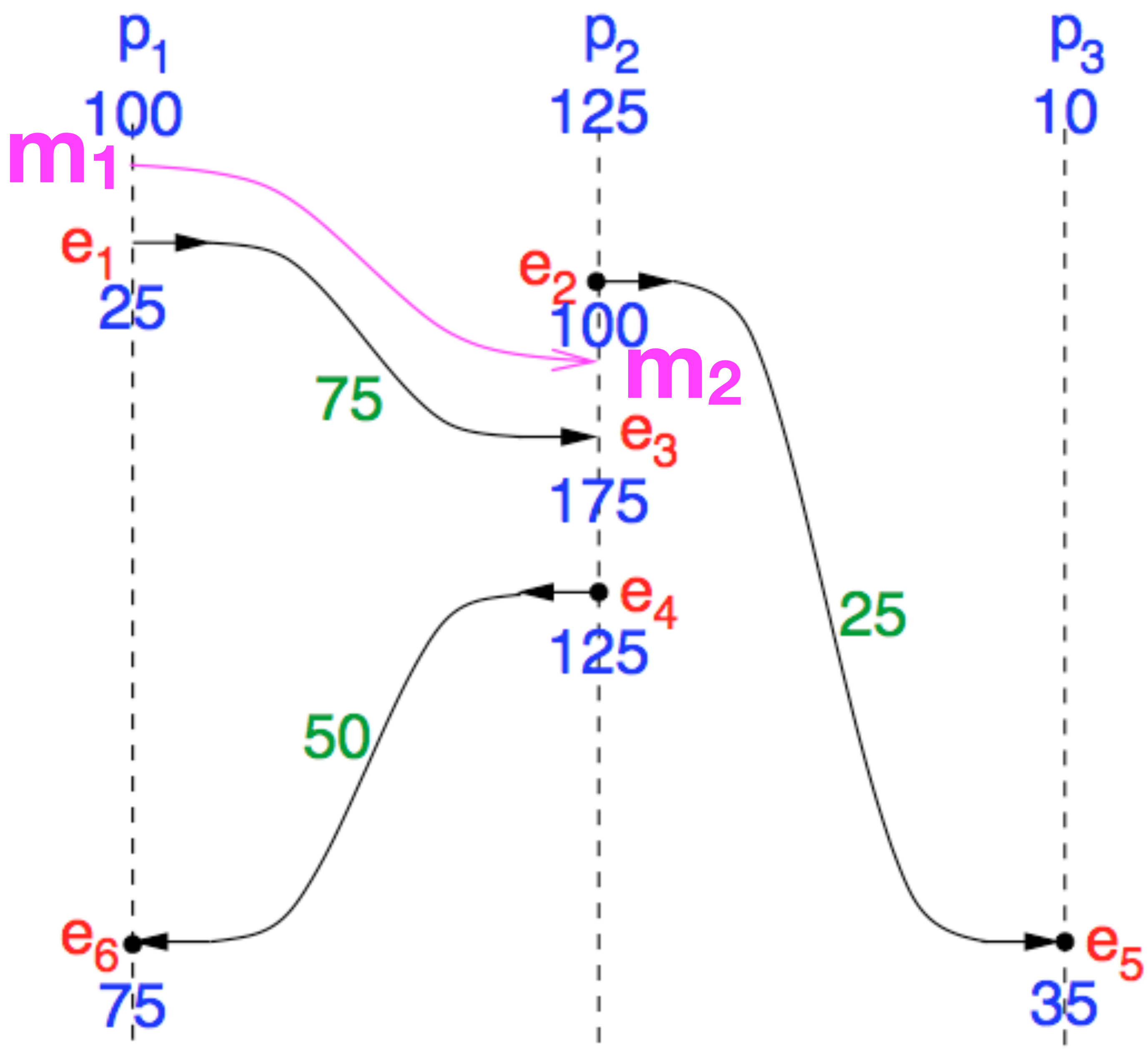
Example 1: The Algorithm In Action...

The computation



Example 1 (cont.): The Algorithm In Action...

p_1 initiates the algorithm

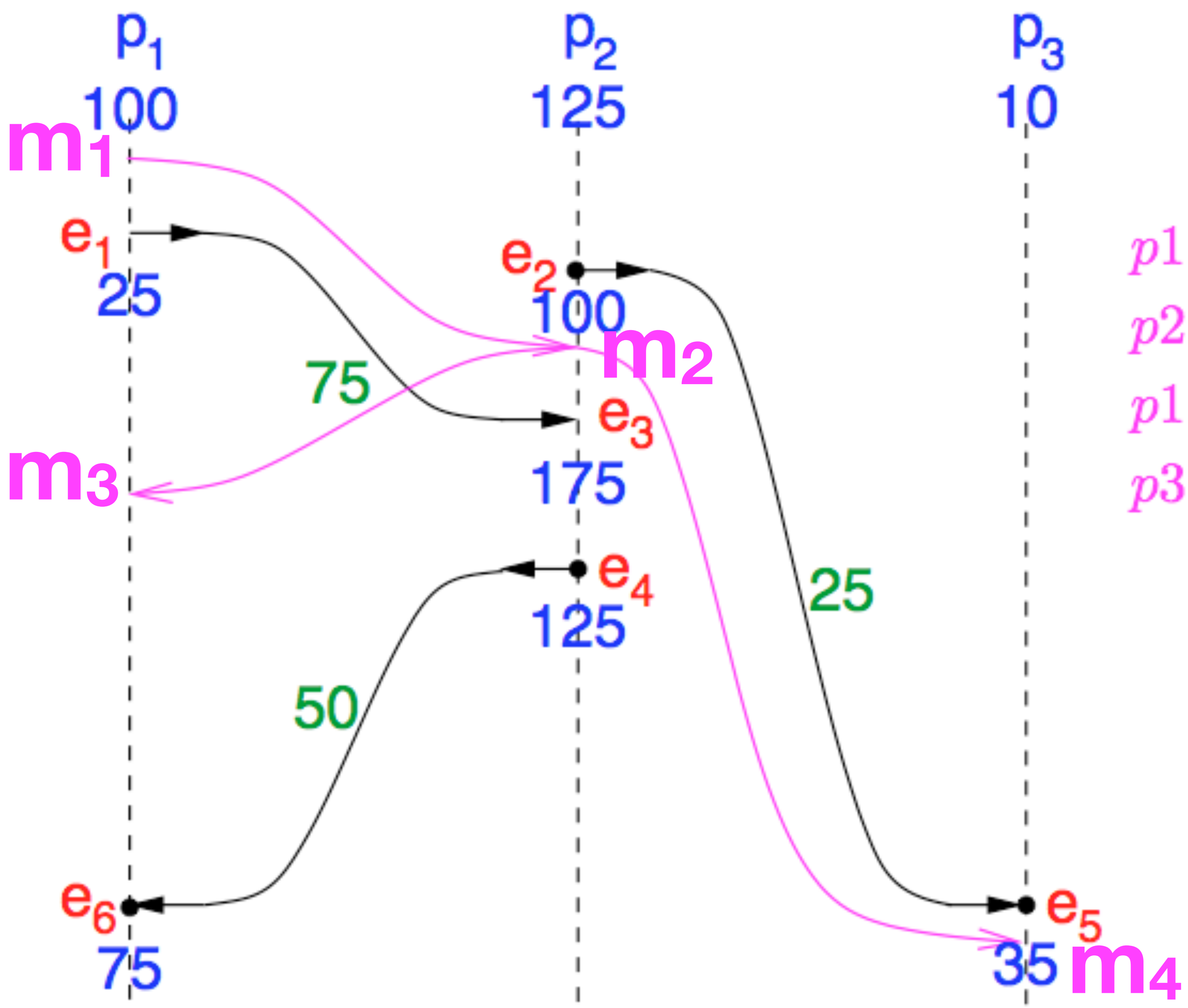


$p1: \quad 100$

$p2: \quad 100 \quad c_{12} = <>$

Example 1 (cont.): The Algorithm In Action...

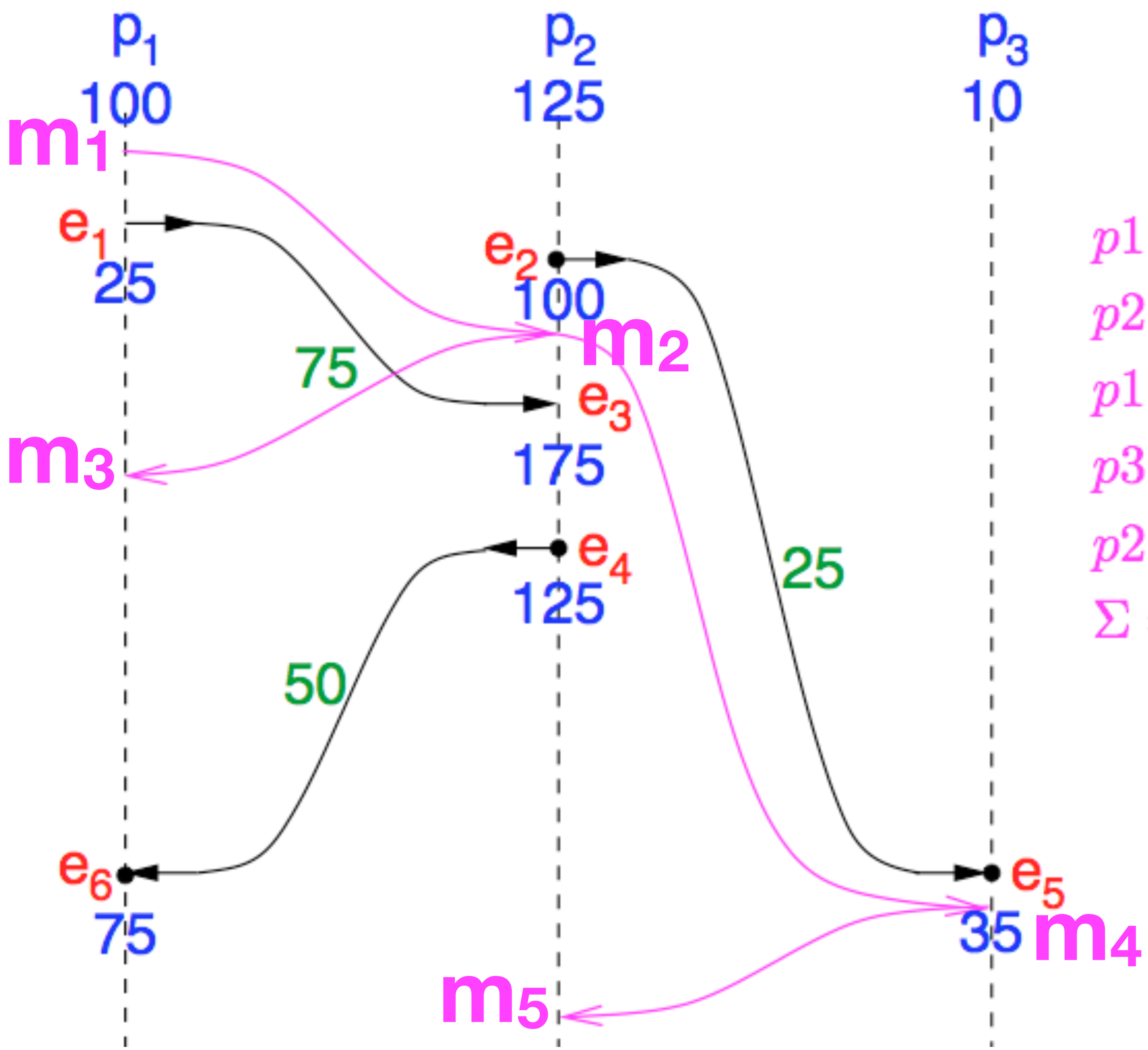
p_1 initiates the algorithm



$$\begin{array}{llll} p1: & 100 & & \\ p2: & 100 & c_{12} = & <> \\ p1: & - & c_{21} = & <> \\ p3: & 35 & c_{23} = & <> \end{array}$$

Example 1 (cont.): The Algorithm In Action...

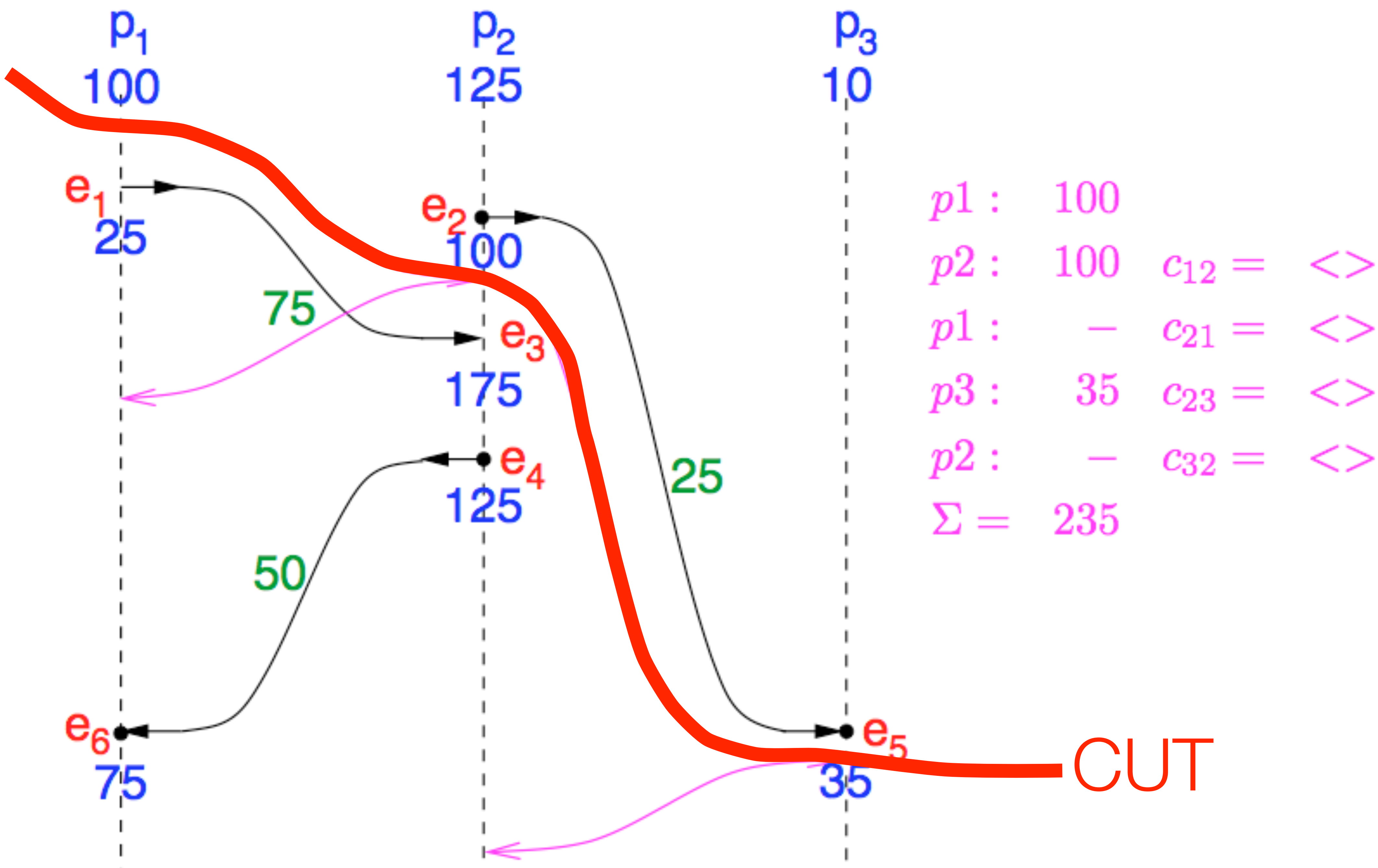
p_1 initiates the algorithm



$$
\begin{array}{llll}
p1: & 100 & & \\
p2: & 100 & c_{12} = & \langle\rangle \\
p1: & - & c_{21} = & \langle\rangle \\
p3: & 35 & c_{23} = & \langle\rangle \\
p2: & - & c_{32} = & \langle\rangle \\
\Sigma = & 235 & &
\end{array}
$$

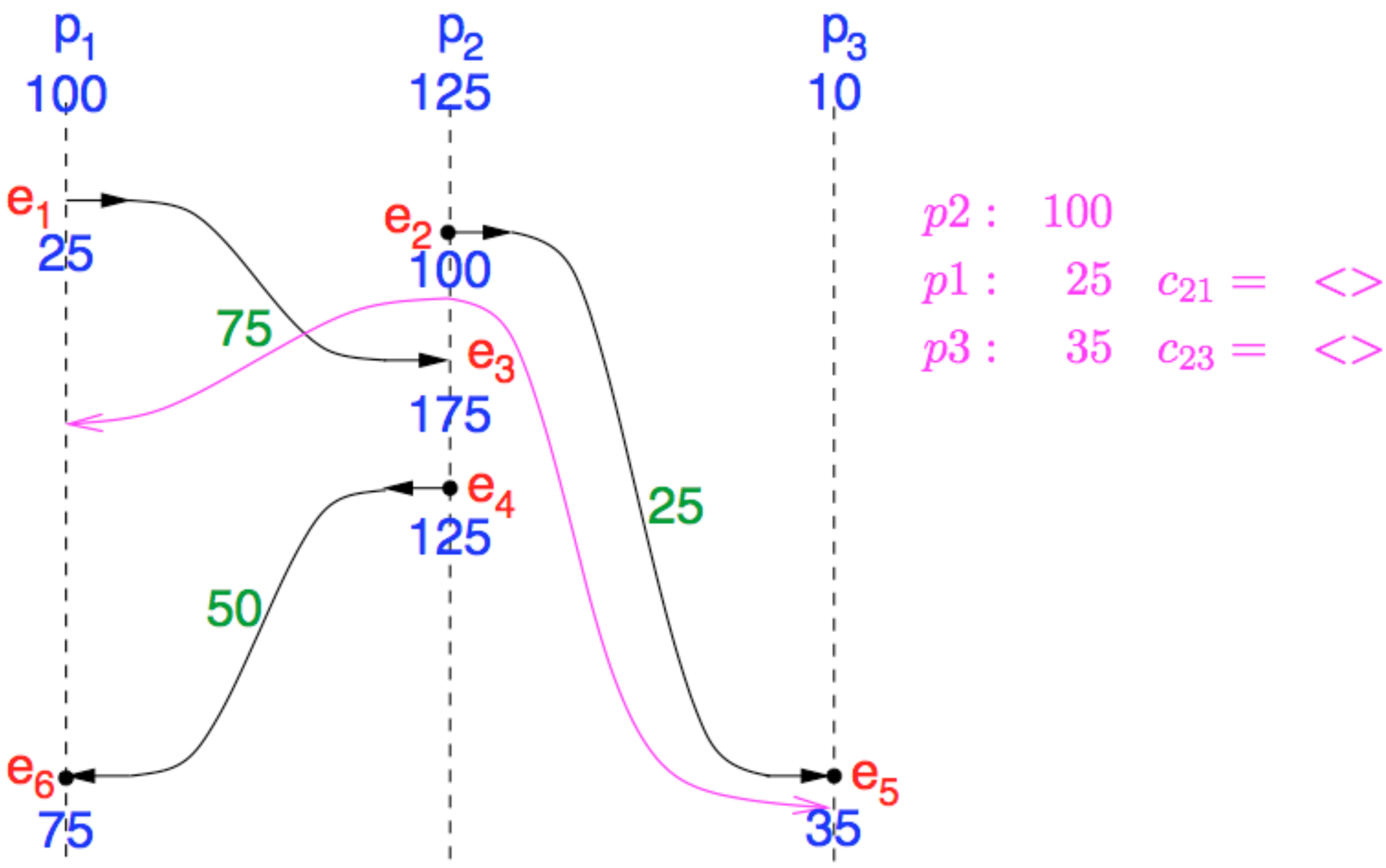
Example 1 (cont.): The Algorithm In Action...

p_1 initiates the algorithm



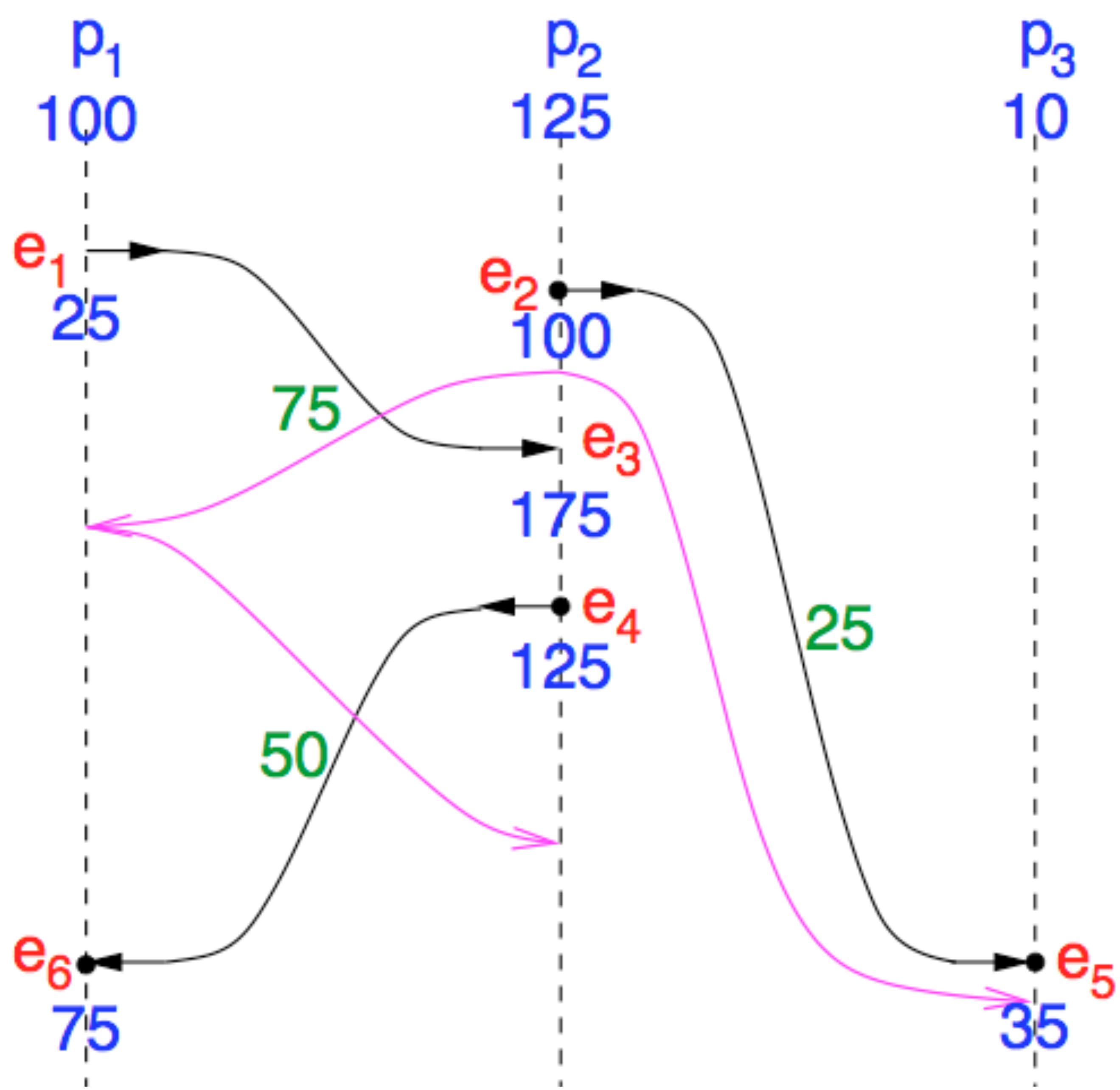
Example 2: The Algorithm In Action...

p_2 initiates the algorithm



Example 2 (cont.): The Algorithm In Action...

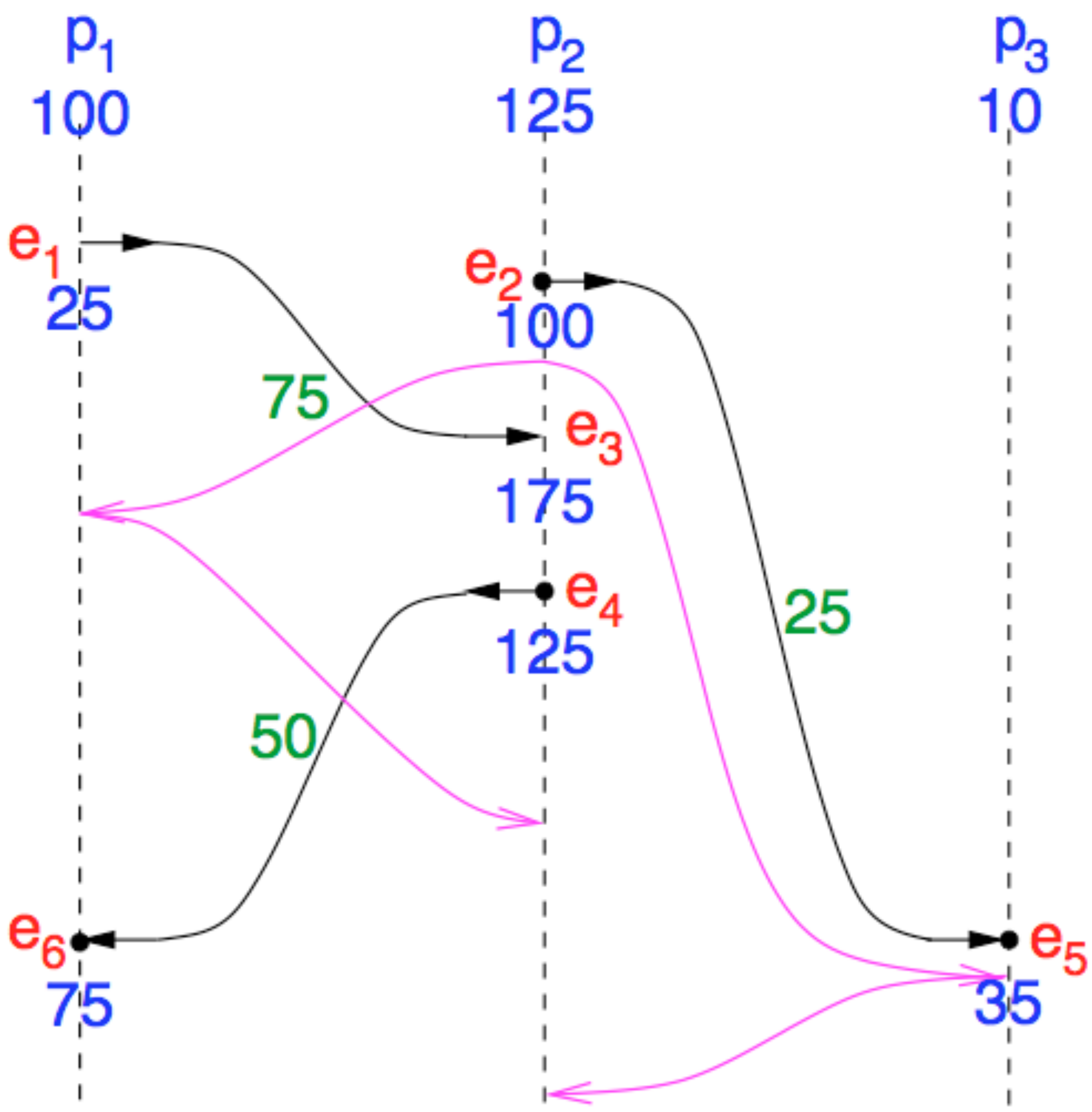
p_2 initiates the algorithm



$$p2: \quad 100$$
$$p1: \quad 25 \quad c_{21} = \ <>$$
$$p3: \quad 35 \quad c_{23} = \ <>$$
$$p2: \quad - \quad c_{12} = \langle 75 \rangle$$

Example 2 (cont.): The Algorithm In Action...

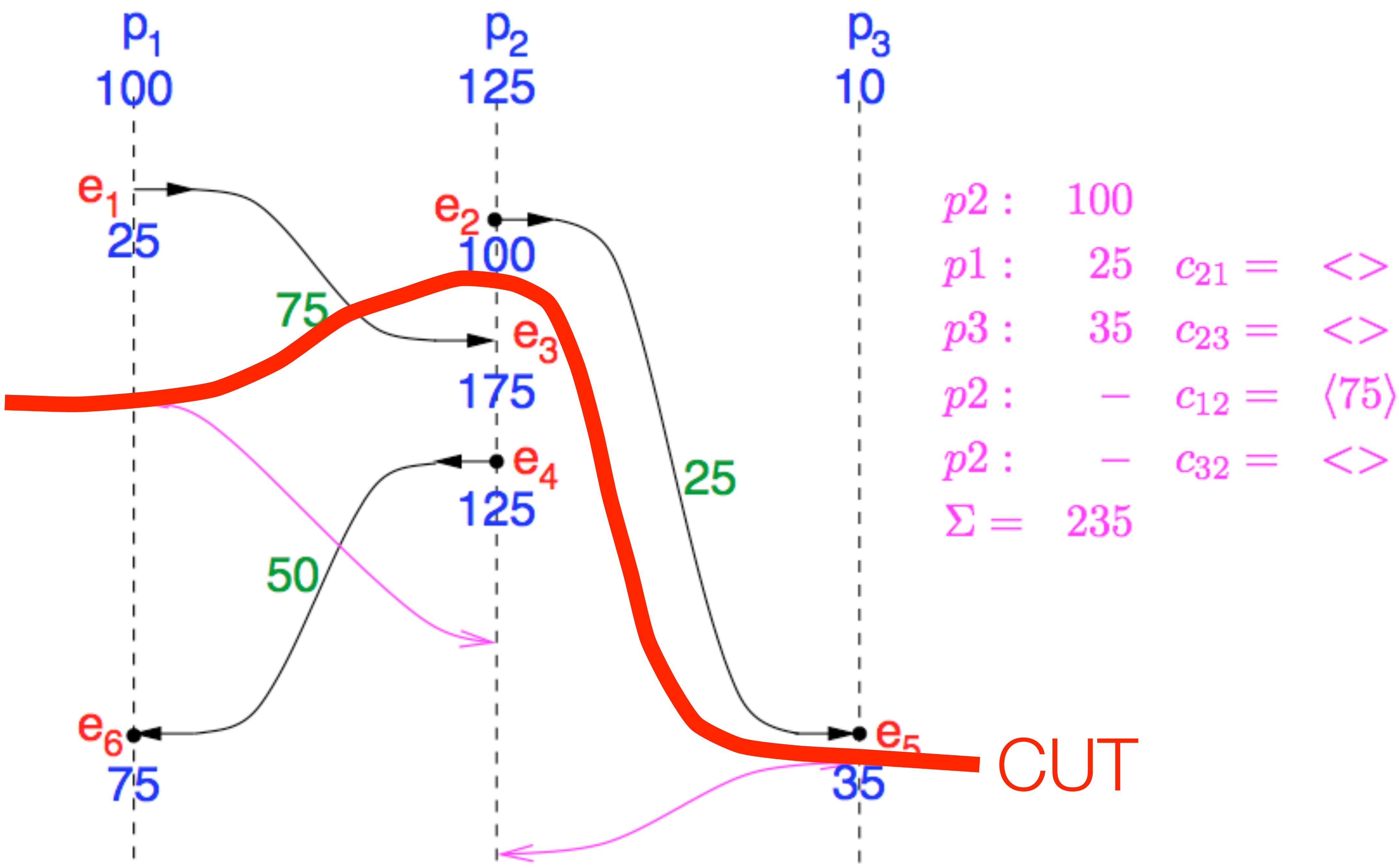
p_2 initiates the algorithm



$$p2: \quad 100$$
$$p1: \quad 25 \quad c_{21} = \langle\rangle$$
$$p3: \quad 35 \quad c_{23} = \langle\rangle$$
$$p2: \quad - \quad c_{12} = \langle 75 \rangle$$
$$p2: \quad - \quad c_{32} = \langle\rangle$$
$$\Sigma = \quad 235$$

Example 2 (cont.): The Algorithm In Action...

p_2 initiates the algorithm



How the Global Snapshot is Then Collected?

- In a practical implementation, the recorded local snapshots must be put together to create a global snapshot of the distributed system

- How? Several policies:

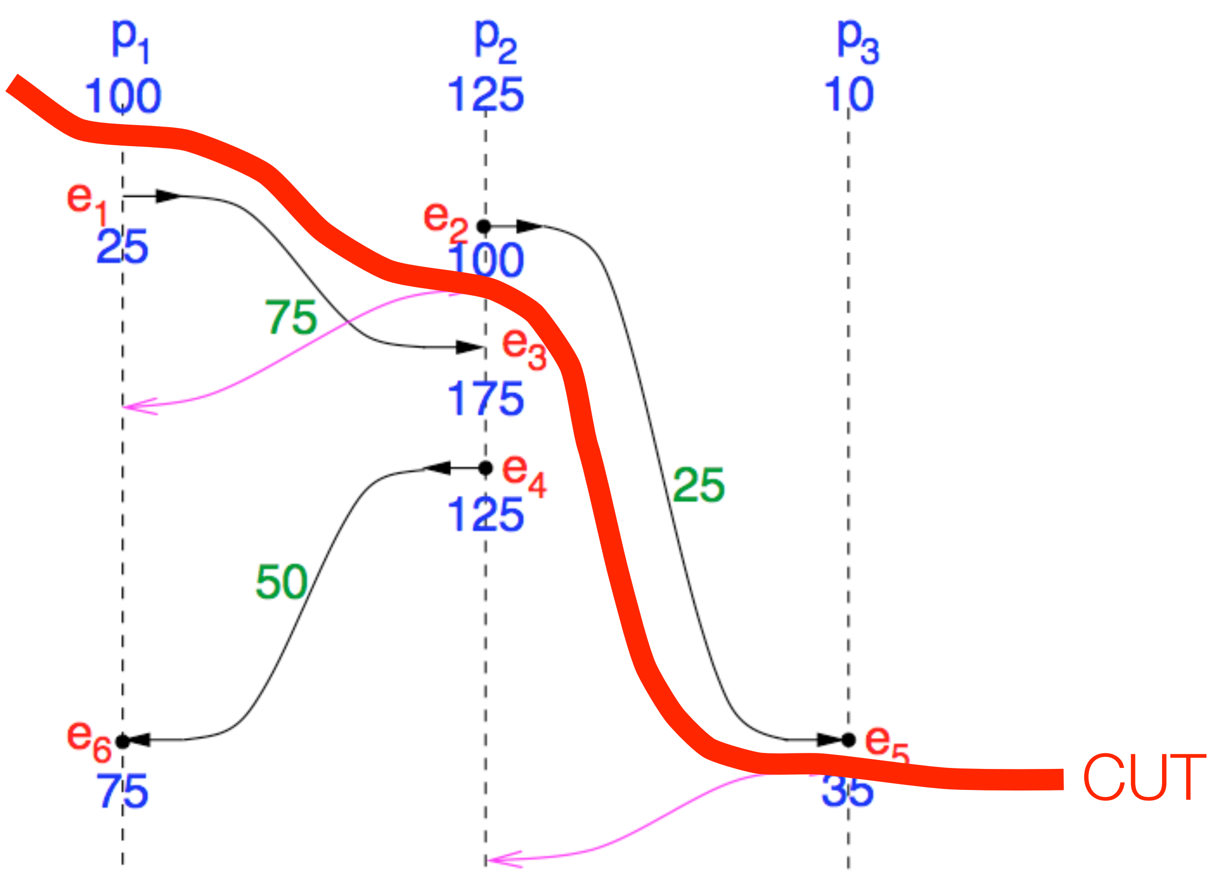
  - each process sends its local snapshot to the initiator of the algorithm

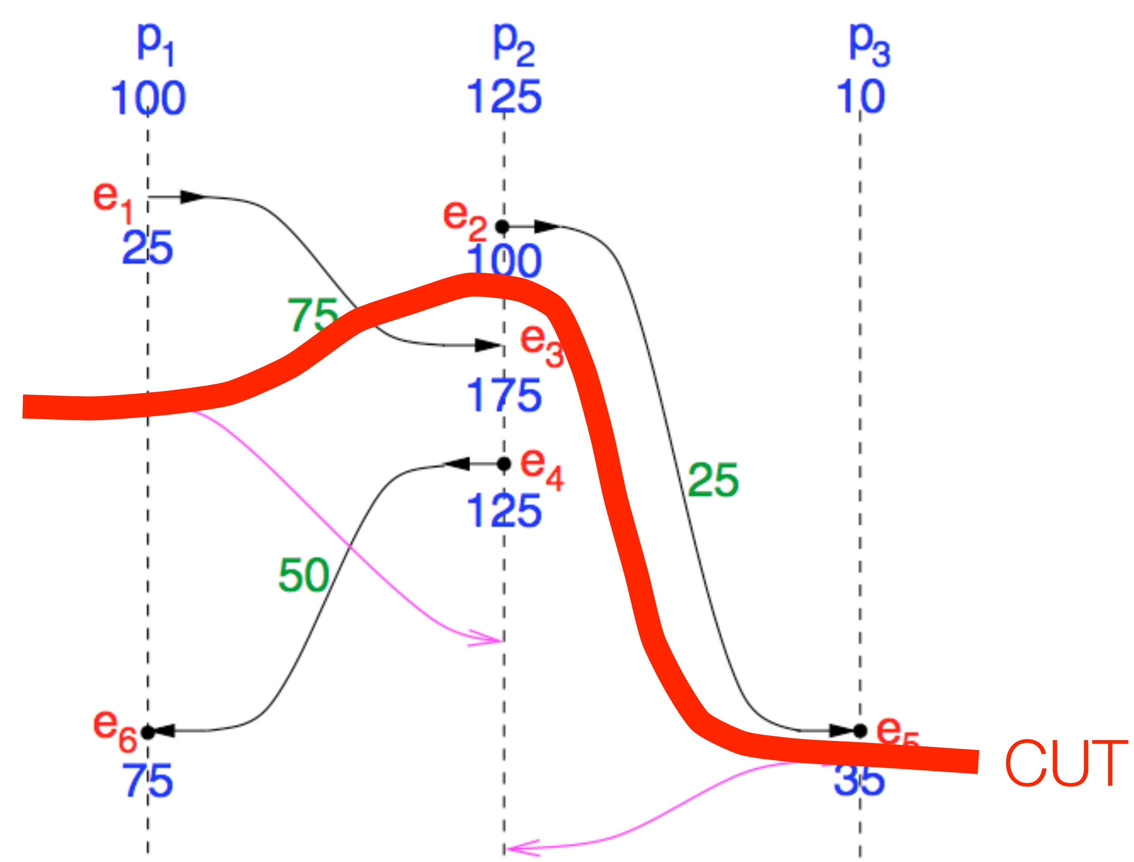
  - each process sends the information it records along all outgoing channels and each process receiving such information for the first time propagates it along its outgoing channels

Complexity of the Snapshot Algorithm

- The recording part of a single instance of the algorithm requires:
  - *O(e)* messages, where *e* is the number of edges in the network
  - *O(d)* time, where *d* is the diameter of the network
- Diameter of a network: the longest of all the shortest paths in a network

How is That Possible??!!





$p1:$	100	
$p2:$	100	$c_{12} = <>$
$p1:$	$-$	$c_{21} = <>$
$p3:$	35	$c_{23} = <>$
$p2:$	$-$	$c_{32} = <>$
$\Sigma =$	235	

In both these possible runs of the algorithm, the recorded global states **NEVER** occurred in the actual execution!

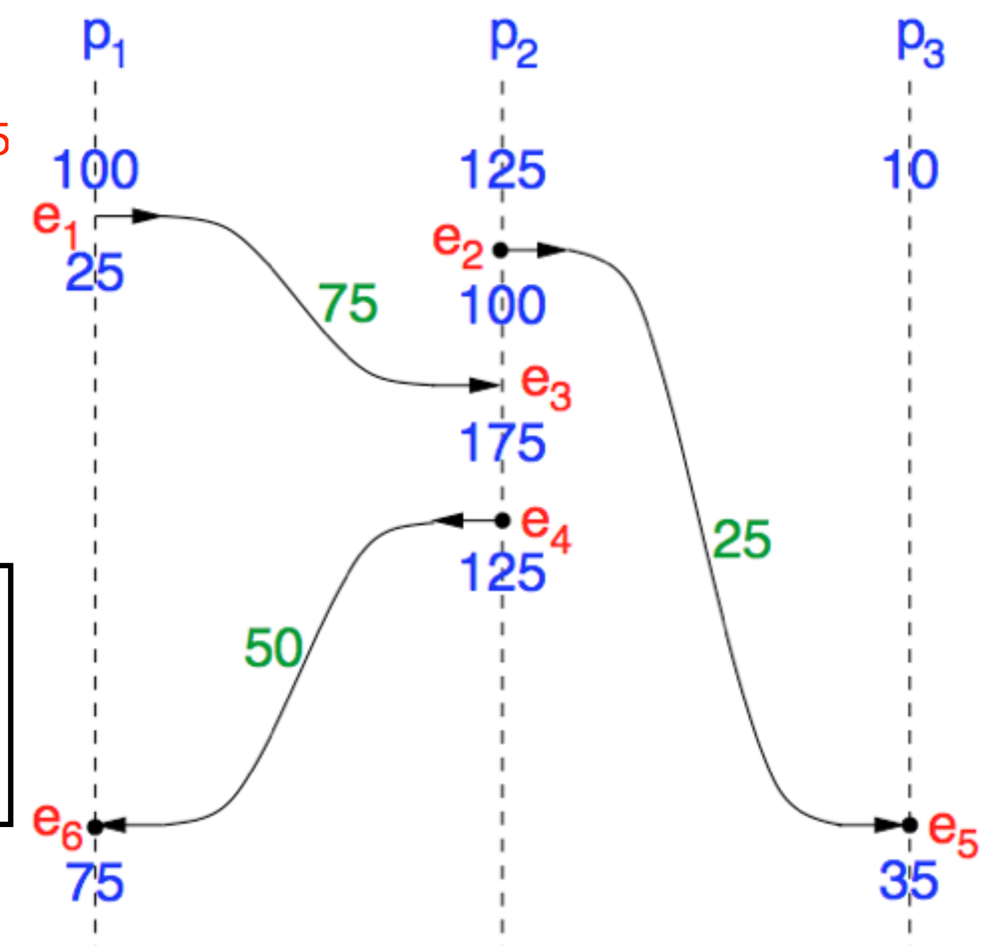
$p2:$	100	
$p1:$	25	$c_{21} = <>$
$p3:$	35	$c_{23} = <>$
$p2:$	$-$	$c_{12} = \langle 75 \rangle$
$p2:$	$-$	$c_{32} = <>$
$\Sigma =$	235	

Incomparable Events!

- The algorithm finds a global state based on a *partial ordering* → of events.

For instance, we know that $e_2 \to e_3$ and $e_2 \to e_5$ BUT we have no knowledge about the timing relationship of e_3 and e_5.
With respect to →, e_3 and e_5 are *incomparable*!

We cannot determine what the ***true sequence*** of these events is!



- *When we record a process' state, we are unable to know whether the events which we have already seen in this process lay before or after incomparable events in other processes*

So... What Does the Algorithm Find?

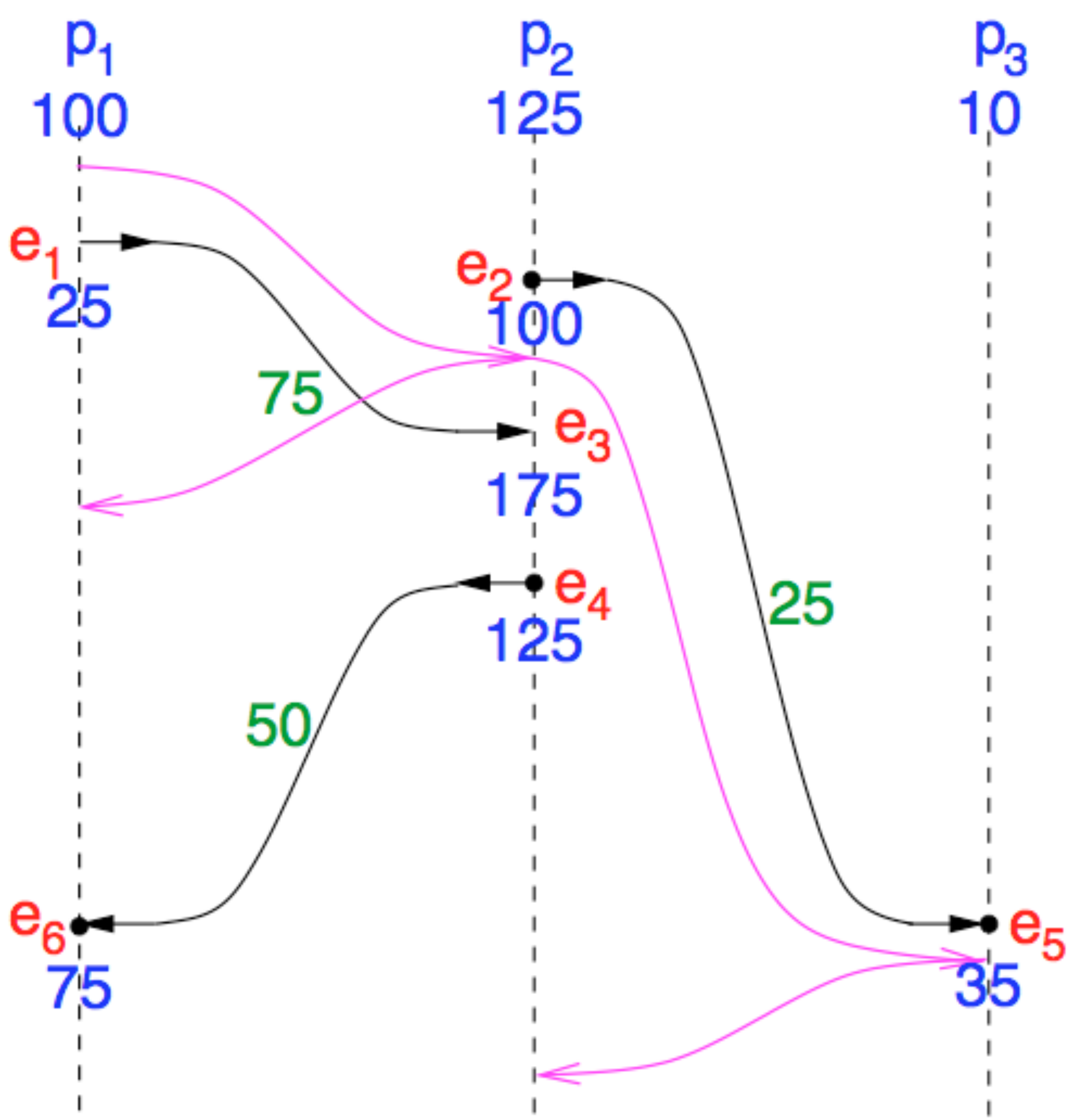
- Pre-recording events: events in a computation which take place BEFORE the process in which they occur records its own state

- Post-recording events: all other events

- *The algorithm finds a global state which corresponds to a PERMUTATION of the actual order of the events, such that all pre-recording events come before all post-recording events*

- The recorded global state, S*, is the one which would be found after all the pre-recording events and before all the post-recording events

Example



$$
\begin{array}{llll}
p1: & 100 & & \\
p2: & 100 & c_{12} = & <> \\
p1: & - & c_{21} = & <> \\
p3: & 35 & c_{23} = & <> \\
p2: & - & c_{32} = & <> \\
\Sigma = & 235 & &
\end{array}
$$

pre-recording events: $\{e_2, e_5\}$

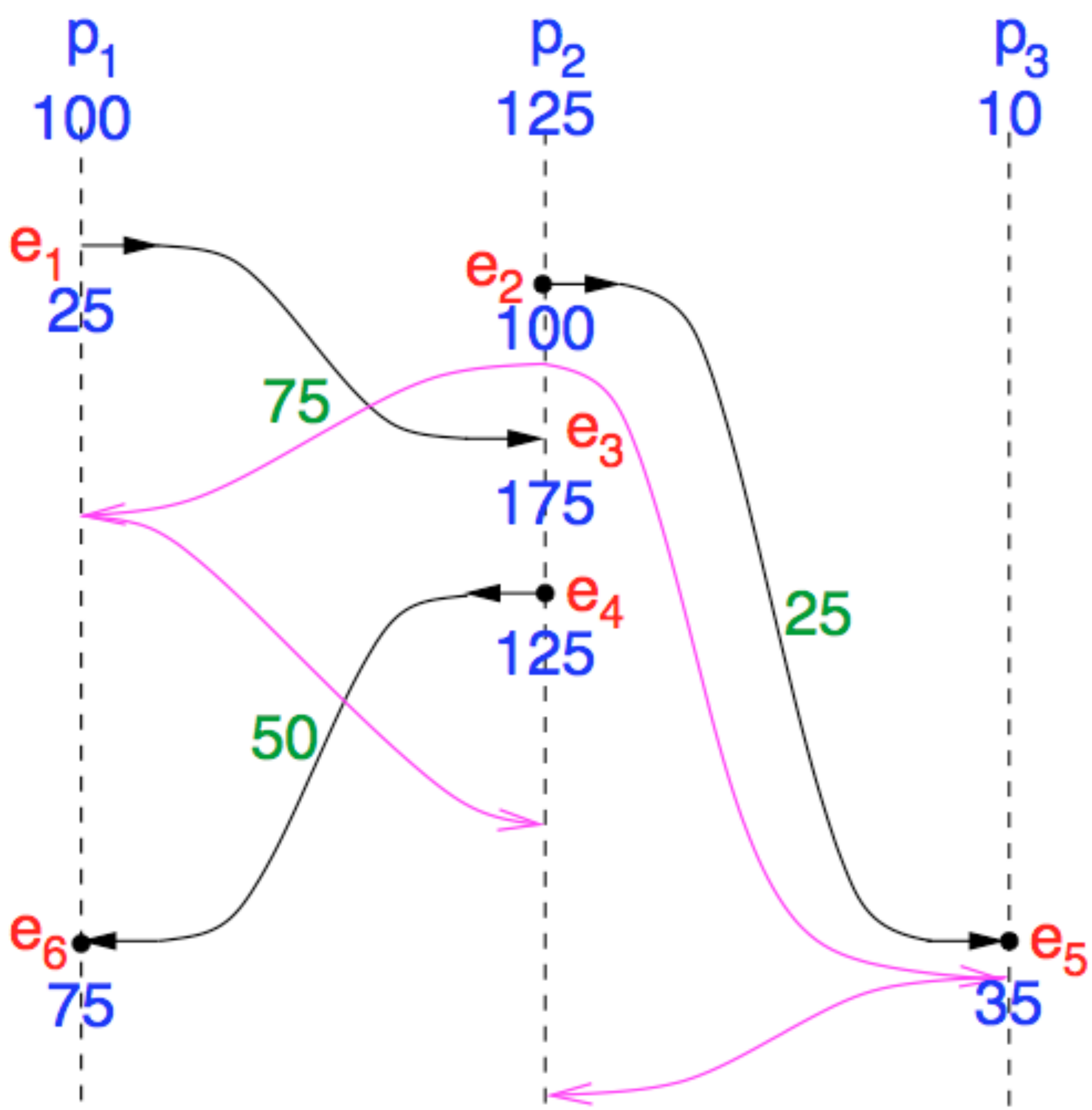
$$seq = \langle e_1, e_2, e_3, e_4, e_5, e_6 \rangle$$

$$seq' = \langle e_2, e_5 | e_1, e_3, e_4, e_6 \rangle$$

S^* recorded global state

Course 02222, DTU, Spring 2009. – r

Example



$p2:$	100		
$p1:$	25	$c_{21} =$	$<>$
$p3:$	35	$c_{23} =$	$<>$
$p2:$	$-$	$c_{12} =$	$\langle 75 \rangle$
$p2:$	$-$	$c_{32} =$	$<>$
$\Sigma =$	235		

pre-recording events: $\{e_1, e_2, e_5\}$

$$seq = \langle e_1, e_2, e_3, e_4, e_5, e_6 \rangle$$

$$seq' = \langle e_1, e_2, e_5 \mid e_3, e_4, e_6 \rangle$$

S^* recorded global state

Global State Could Possibly Have Occurred!

- S^* is a state which *could possibly have occurred*, in the sense that:
  - It is possible to reach S^* via a sequence of *possible events* starting from the initial state of the system, S_i (in the previous example: <e_1, e_2, e_5>)
  - It is possible to reach the final state of the system, S_f, via a sequence of possible events starting from S^* (in the previous example: <e_3, e_4, e_6>)

$$seq' = \langle e_1, e_2, e_5 \mid e_3, e_4, e_6 \rangle$$

S^* recorded global state

Oh Man... So Why Recording Global State?

- Stable property: a property that persists, such as termination or deadlock
- Idea: if a stable property holds in the system before the snapshot begins, it holds in the recorded global snapshot
- A recorded global state is useful in **DETECTING STABLE PROPERTIES**
- Examples:
  - Failure recovery: a global state (checkpoint) is periodically saved and recovery from a process failure is done by restoring the system to the last saved global state
  - Debugging: the system is restored to a consistent global state and the execution resumes from there in a controlled manner

Outline

- Global State - *what is a global state of a distributed system?*
  - definition
  - next global state
- Distributed Snapshot - *how to record a global state of a distributed system?*
  - consistent global states
  - Chandy and Lamport's algorithm
- Evaluating Predicates - *why/how to use the recorded global states?*
  - evaluating Stable Predicates
  - evaluating Non Stable Predicates

Stable Predicates

The Problem

- Let **Σ** be a global state built by one of the methods in literature
- It represents a state of the past, that may have no bearing to the present
- Does it make sense to evaluate predicate **Φ** on it?
- A special case: **stable predicates**

  Many systems properties have the characteristics that once they become true, they remain true

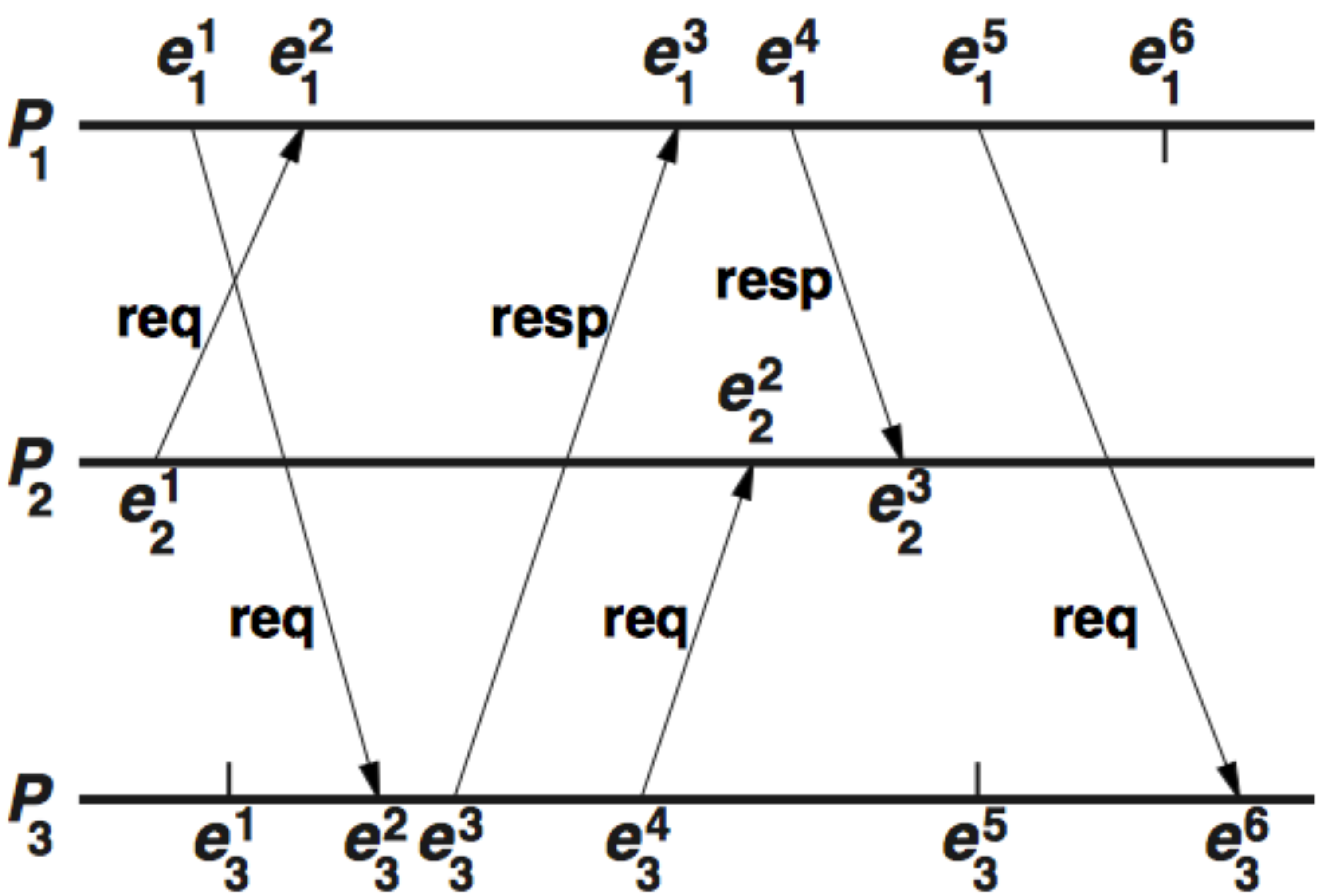
  - Deadlock
  - Garbage collection
  - Termination

Run of a Computation

- A **run** of a computation is a *total ordering* **R** that includes all the events in the global history and that is consistent with each local history
  - In other words, the events of p_i appear in **R** in the same order in which they appear in h_i
  - A run corresponds to the notion that events in a distributed computation *actually* occur in a total order
  - A *distributed computation may correspond to many runs*
- A run **R** is said to be **consistent** if for all events e and e', $e \rightarrow e'$ implies that e appears before e' in **R**

Example of (Consistent) Run

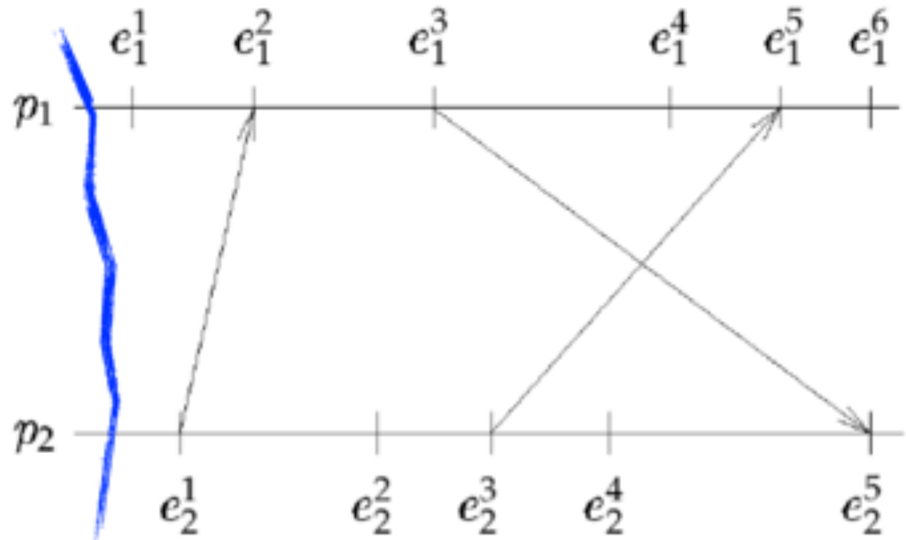
$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$



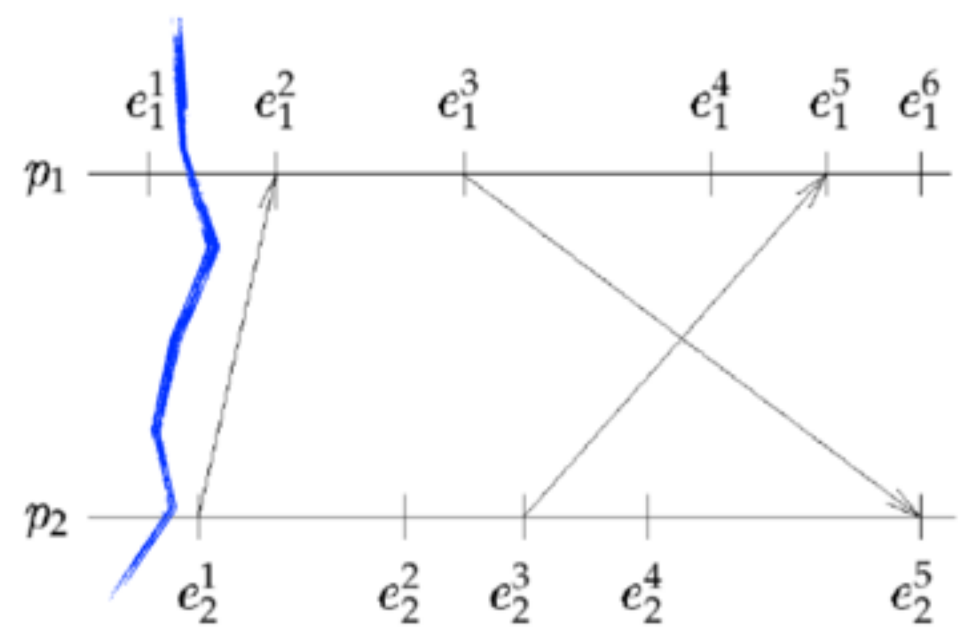
Run as Sequence of Consistent Global States

- A **consistent run R = e^1e^2…** results in a *sequence of* ***consistent*** *global states* **$\Sigma^0\Sigma^1\Sigma^2$…**, where **Σ^0** denotes the initial global state **(σ_1^0, …, σ_n^0)**
- Each (consistent) global state **Σ^i** of the run is obtained from the previous state **Σ^{i-1}** by some process executing the single event ***e^i***

Σ^0



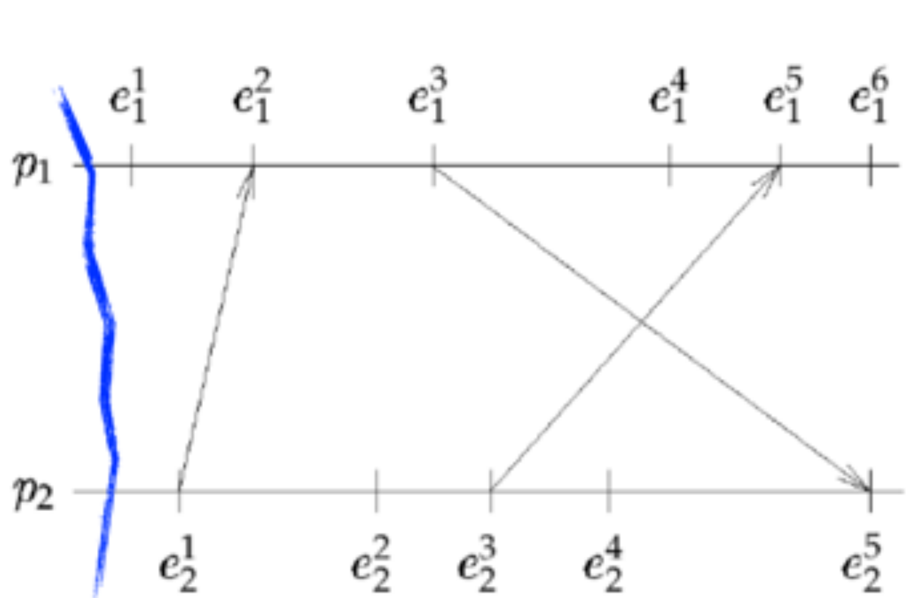
Σ^1



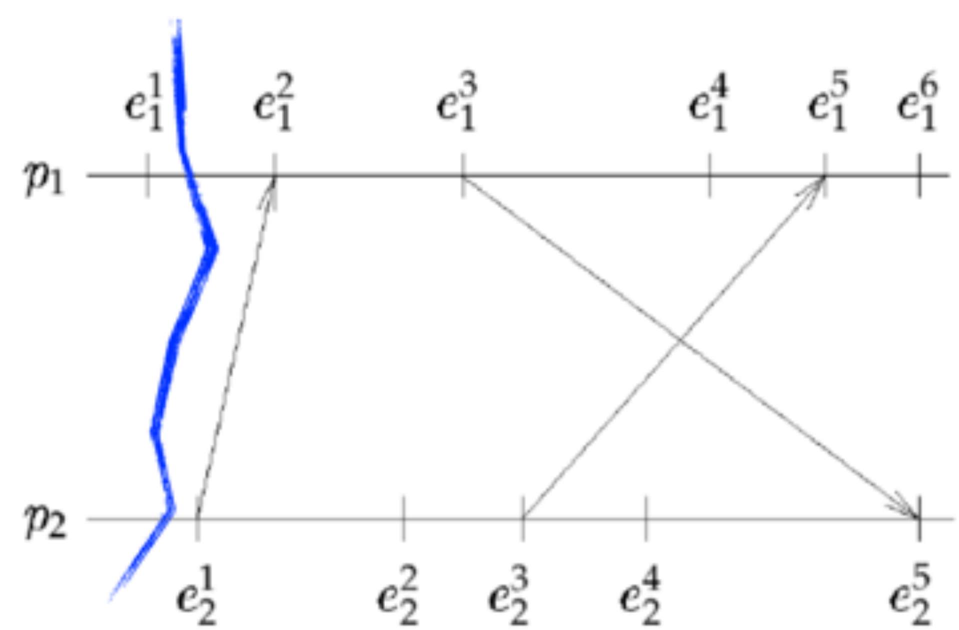
- We use the term **run** to refer to both the sequence of events and the sequence of resulting global states

"Leads To" Relation (⇝)

- For 2 global states of a **consistent** run **R**, we say that a global state **Σ** **leads to** a global state **Σ'** in **R** ($\Sigma \rightsquigarrow_R \Sigma'$) if:
  - **R** results in a sequence of global states $\Sigma^0\Sigma^1\Sigma^2\ldots$
  - $\Sigma = \Sigma^i$, $\Sigma' = \Sigma^j$, $i < j$
- We write $\Sigma \rightsquigarrow \Sigma'$ if there is a run **R** such that $\Sigma \rightsquigarrow_R \Sigma'$

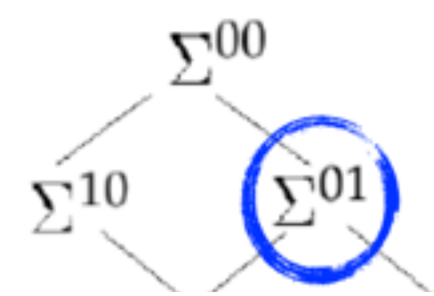


$\Sigma^0 \rightsquigarrow \Sigma^1$



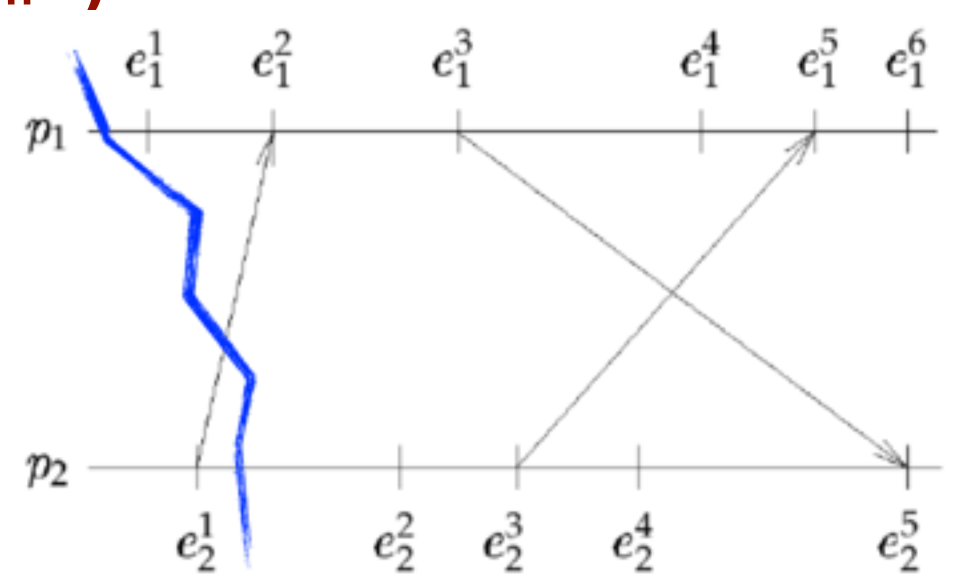
Lattice

- The **set of all *consistent* global states** of a computation along with the **leads-to** relation defines a **lattice**



- n orthogonal axis, one per process

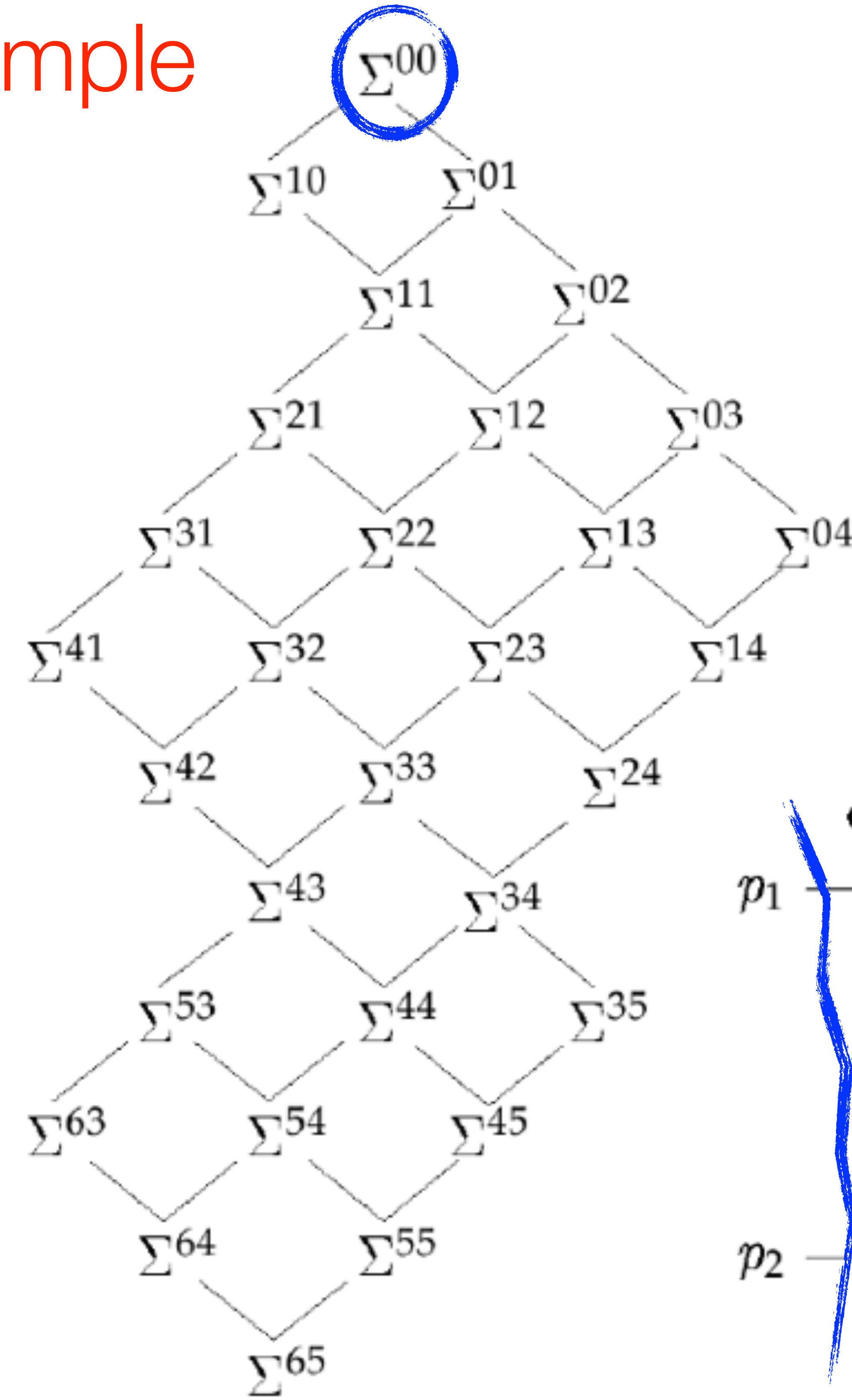
- $\Sigma^{k_1 \ldots k_n}$ shorthand for the global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$
  - Example: n = 2, $\Sigma^{01} = (\sigma_1^{0}, \sigma_2^{1}) = (\varnothing, e_2^{1})$

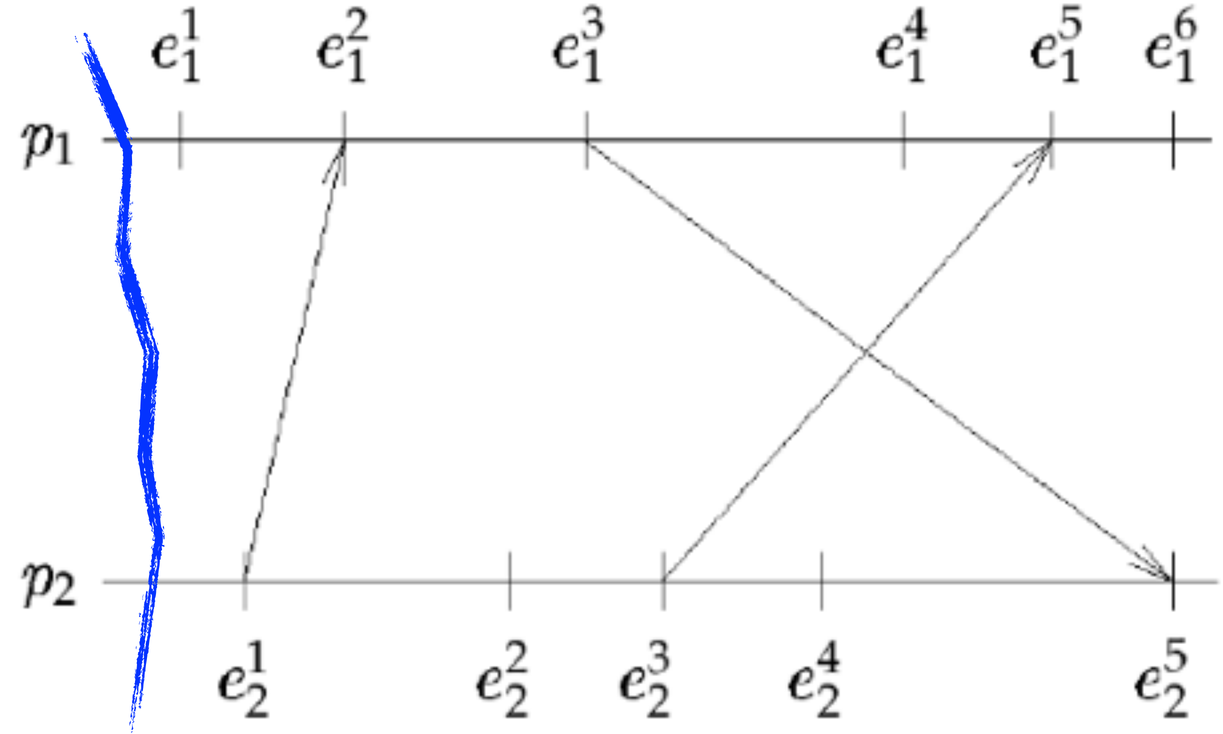


- The **level** of $\Sigma^{k_1 \ldots k_n}$ is equal to $k_1 + \ldots + k_n$

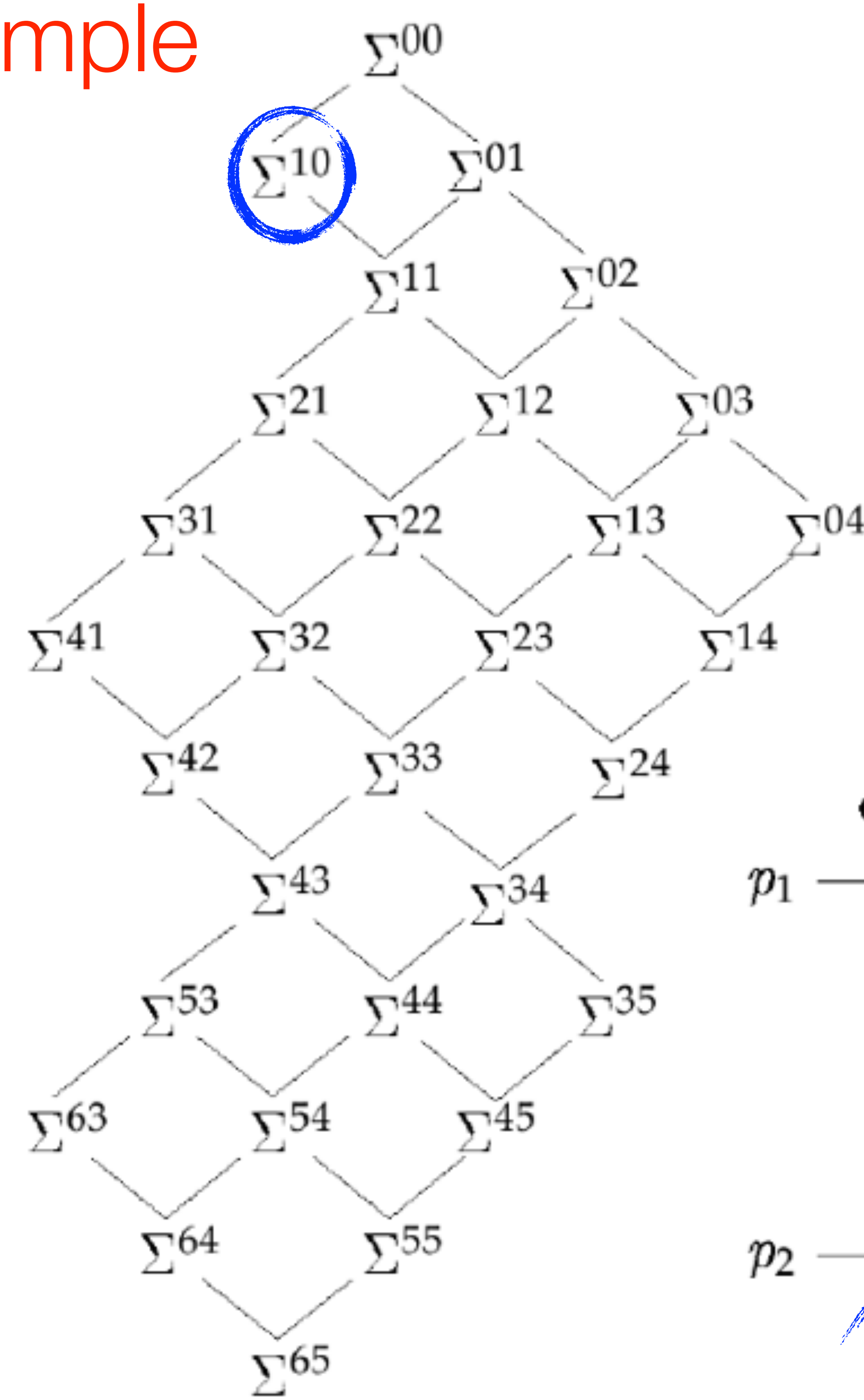
- A **path in the lattice** is a *sequence of global states of increasing levels that corresponds to a **consistent** run*

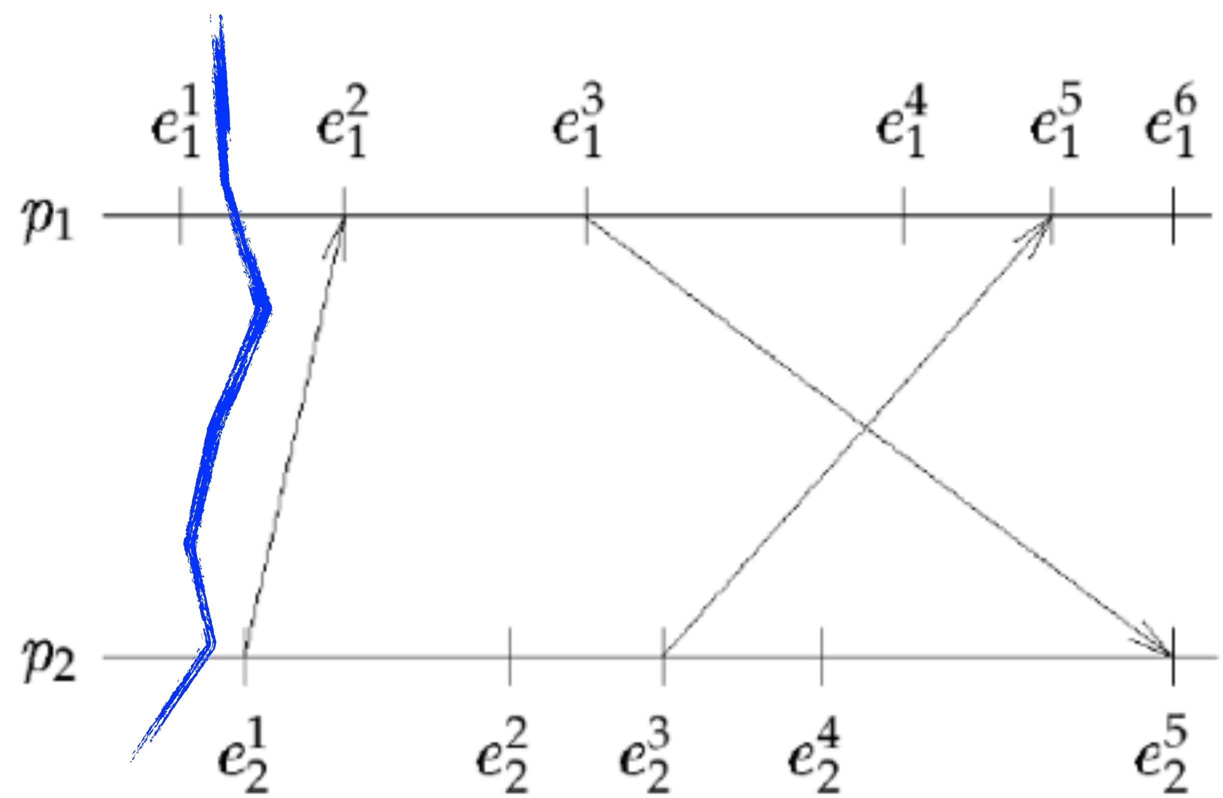
Example


Σ00
Σ10 Σ01
Σ11 Σ02
Σ21 Σ12 Σ03
Σ31 Σ22 Σ13 Σ04
Σ41 Σ32 Σ23 Σ14
Σ42 Σ33 Σ24
Σ43 Σ34
Σ53 Σ44 Σ35
Σ63 Σ54 Σ45
Σ64 Σ55
Σ65

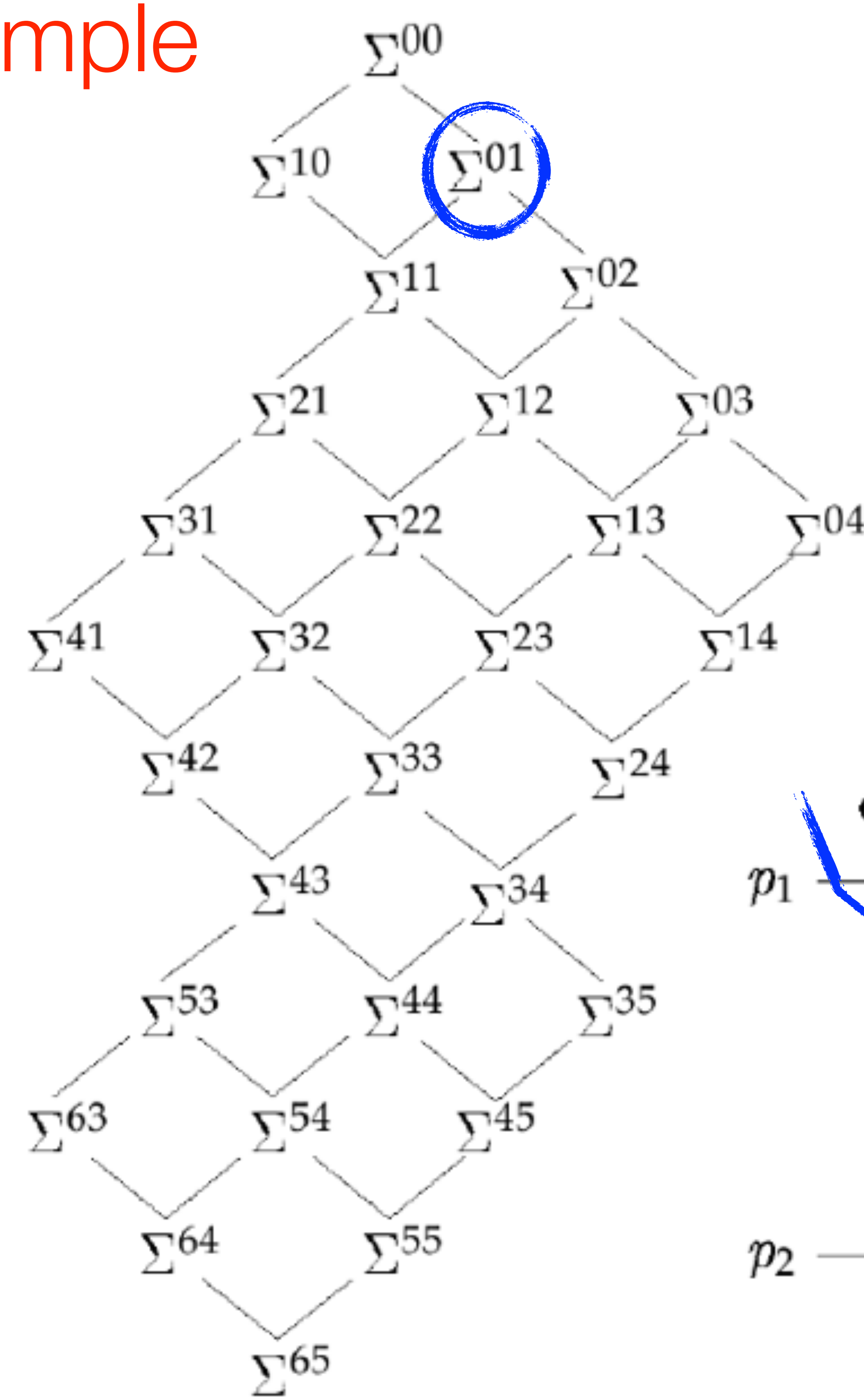

e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
p_1
p_2
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

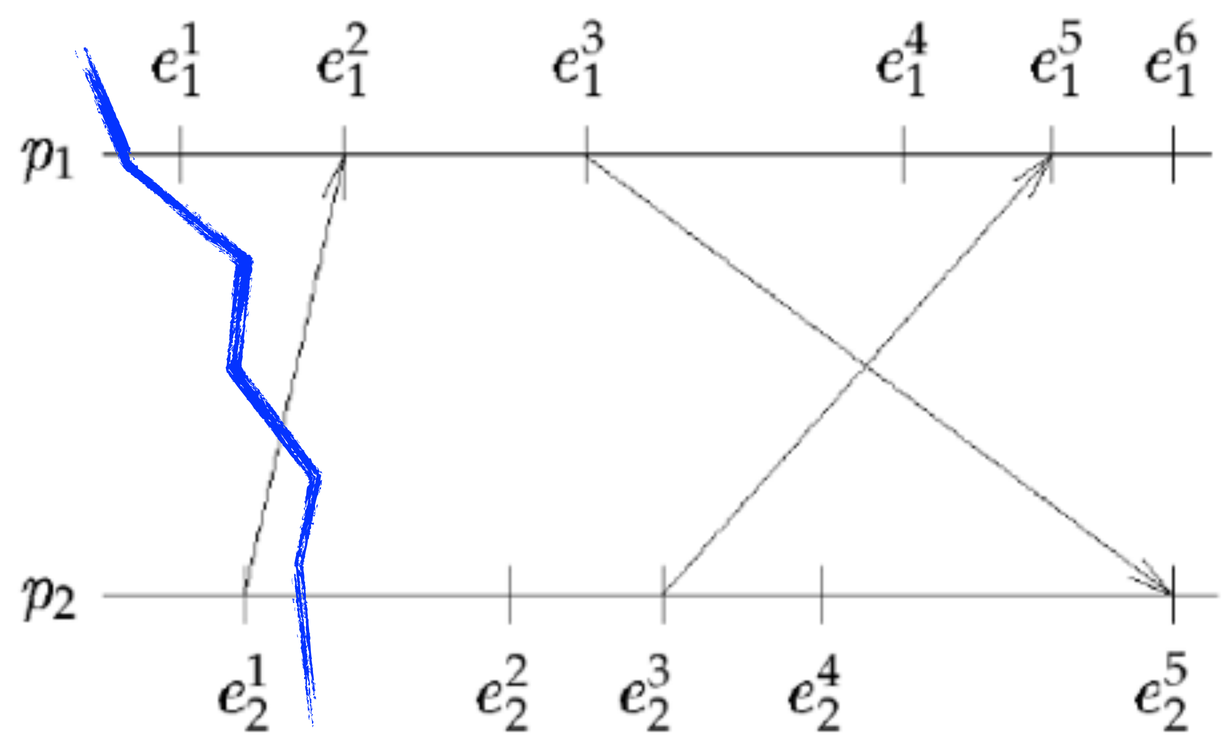
Example


Σ^{00}
Σ^{10} Σ^{01}
Σ^{11} Σ^{02}
Σ^{21} Σ^{12} Σ^{03}
Σ^{31} Σ^{22} Σ^{13} Σ^{04}
Σ^{41} Σ^{32} Σ^{23} Σ^{14}
Σ^{42} Σ^{33} Σ^{24}
Σ^{43} Σ^{34}
Σ^{53} Σ^{44} Σ^{35}
Σ^{63} Σ^{54} Σ^{45}
Σ^{64} Σ^{55}
Σ^{65}

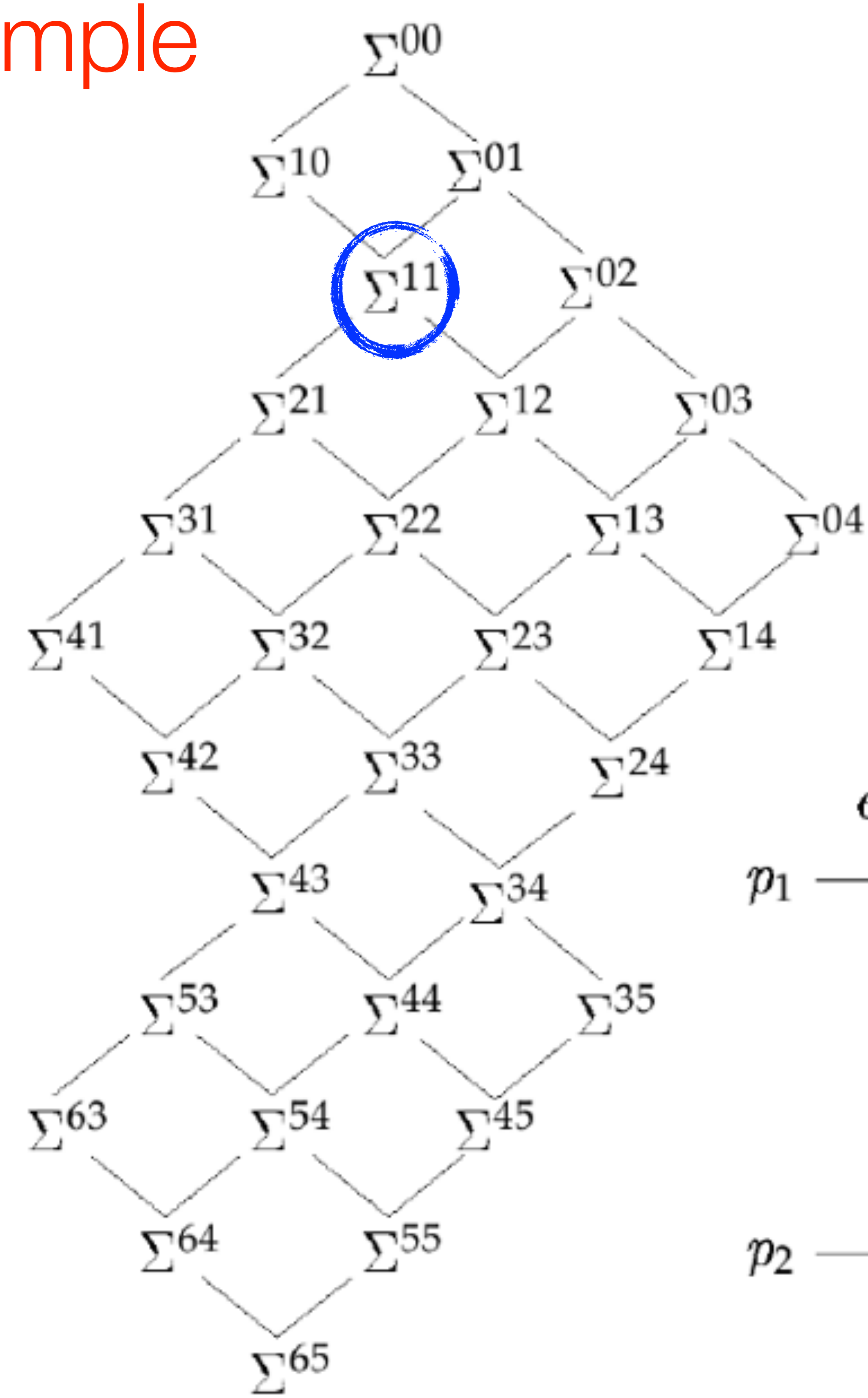

e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
p_1
p_2
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

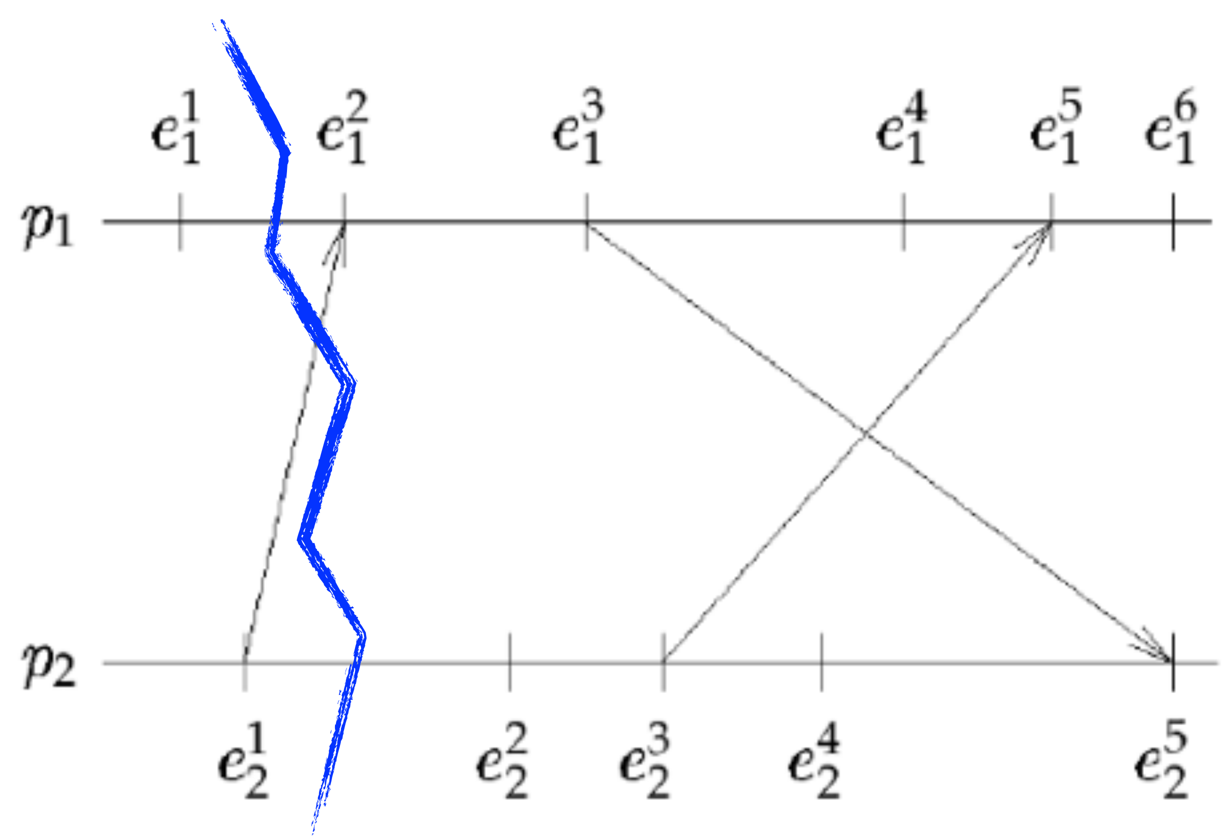
Example


Σ^{00}
Σ^{10} Σ^{01}
Σ^{11} Σ^{02}
Σ^{21} Σ^{12} Σ^{03}
Σ^{31} Σ^{22} Σ^{13} Σ^{04}
Σ^{41} Σ^{32} Σ^{23} Σ^{14}
Σ^{42} Σ^{33} Σ^{24}
Σ^{43} Σ^{34}
Σ^{53} Σ^{44} Σ^{35}
Σ^{63} Σ^{54} Σ^{45}
Σ^{64} Σ^{55}
Σ^{65}

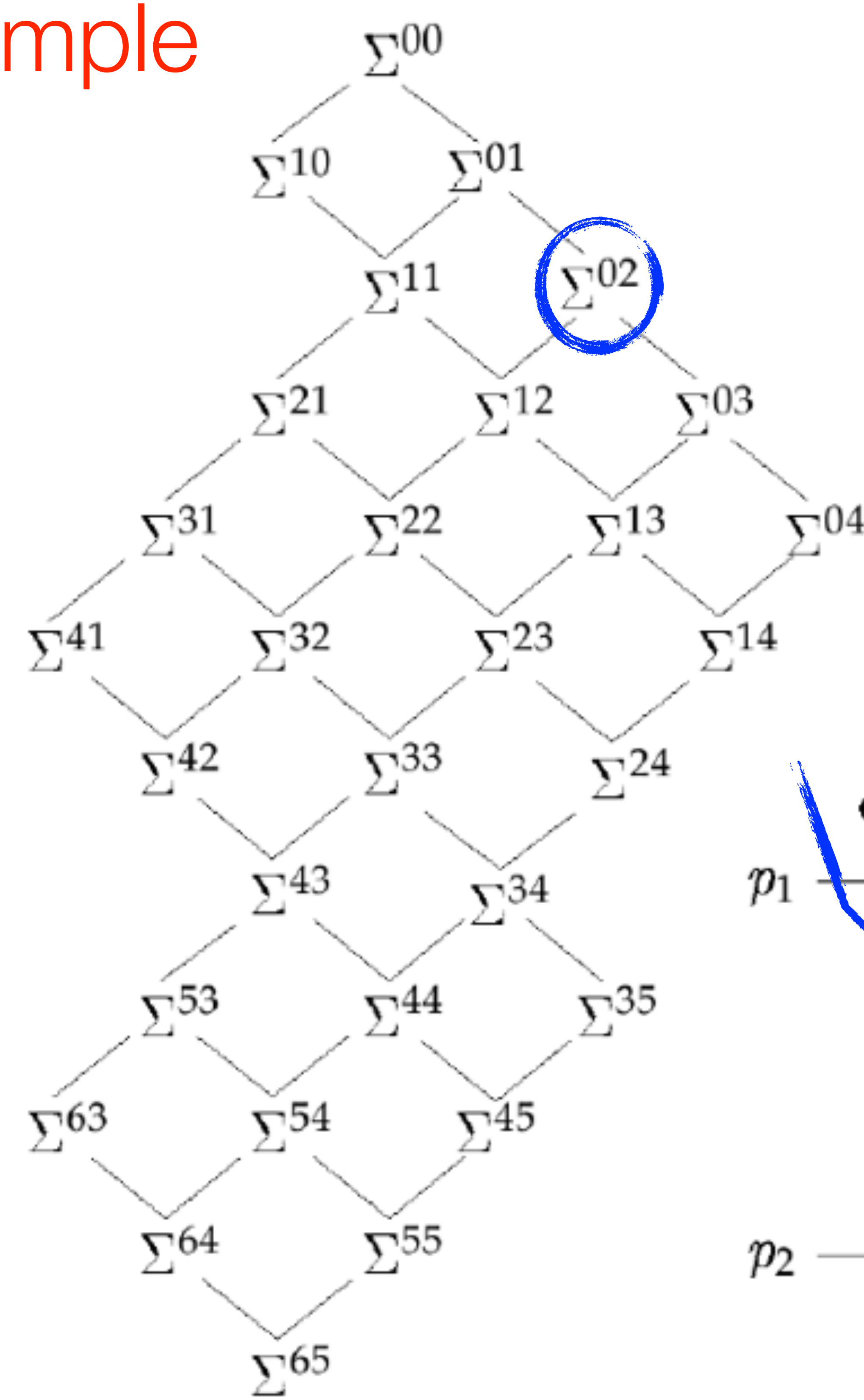

e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
p_1
p_2
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

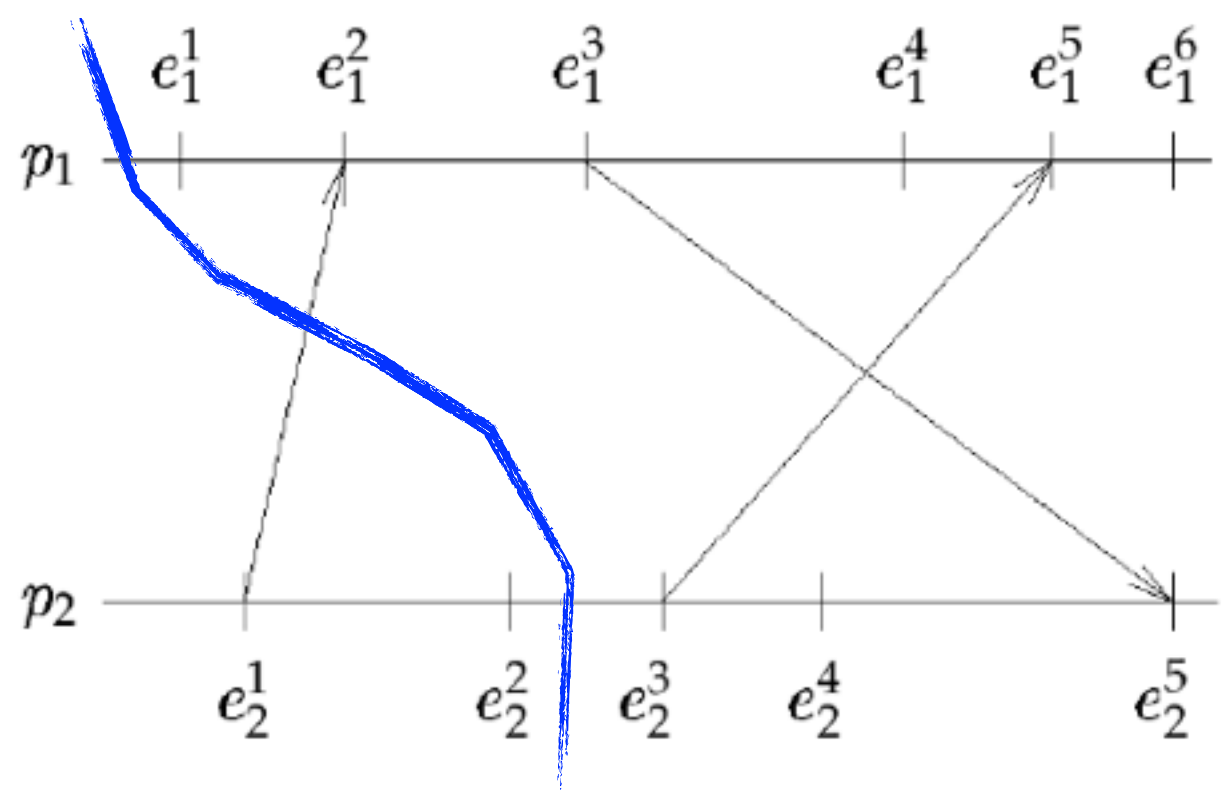
Example


Σ00
Σ10
Σ01
Σ11
Σ02
Σ21
Σ12
Σ03
Σ31
Σ22
Σ13
Σ04
Σ41
Σ32
Σ23
Σ14
Σ42
Σ33
Σ24
Σ43
Σ34
Σ53
Σ44
Σ35
Σ63
Σ54
Σ45
Σ64
Σ55
Σ65

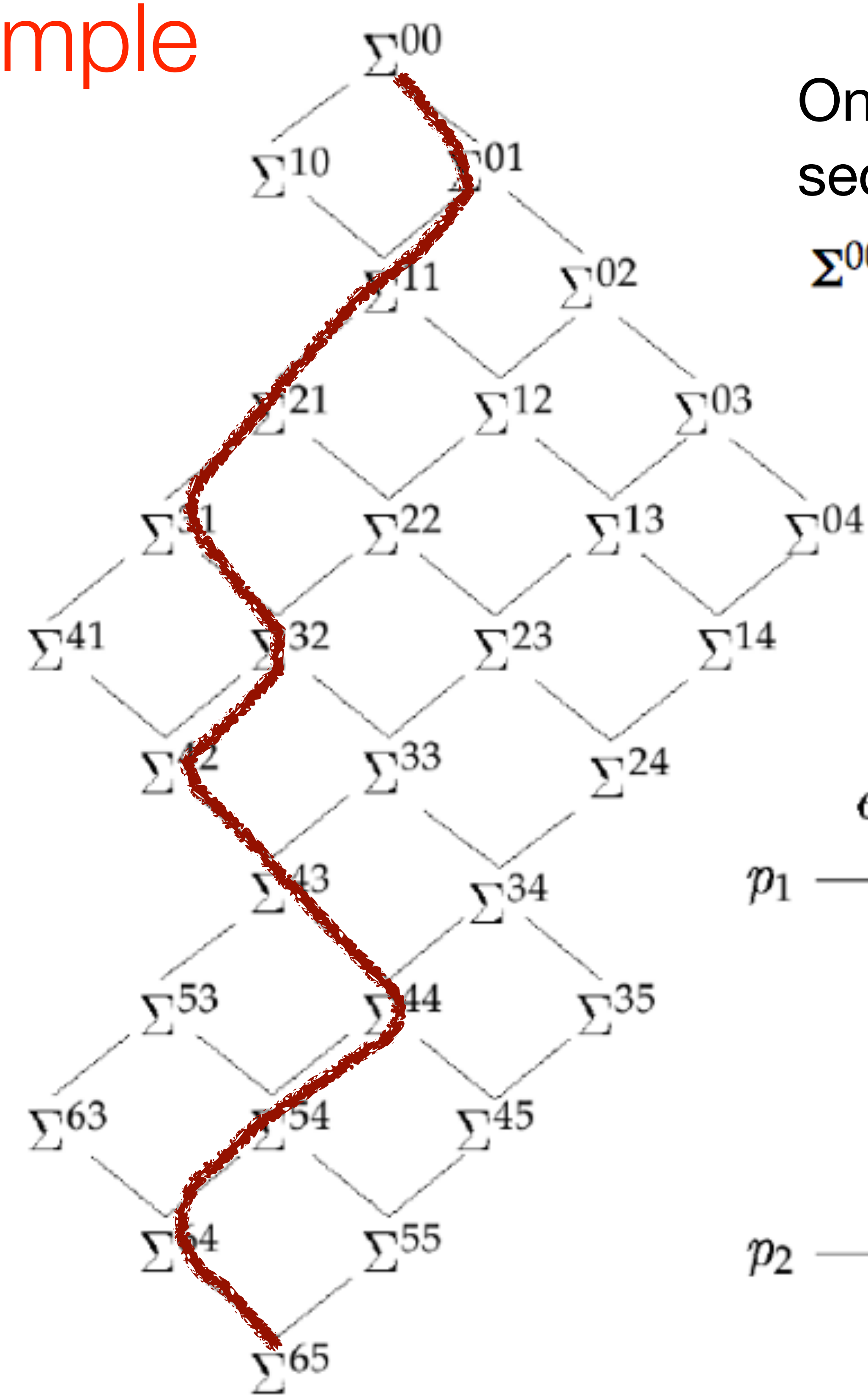

e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
p_1
p_2
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

Example


Σ^{00}
Σ^{10} Σ^{01}
Σ^{11} Σ^{02}
Σ^{21} Σ^{12} Σ^{03}
Σ^{31} Σ^{22} Σ^{13} Σ^{04}
Σ^{41} Σ^{32} Σ^{23} Σ^{14}
Σ^{42} Σ^{33} Σ^{24}
Σ^{43} Σ^{34}
Σ^{53} Σ^{44} Σ^{35}
Σ^{63} Σ^{54} Σ^{45}
Σ^{64} Σ^{55}
Σ^{65}

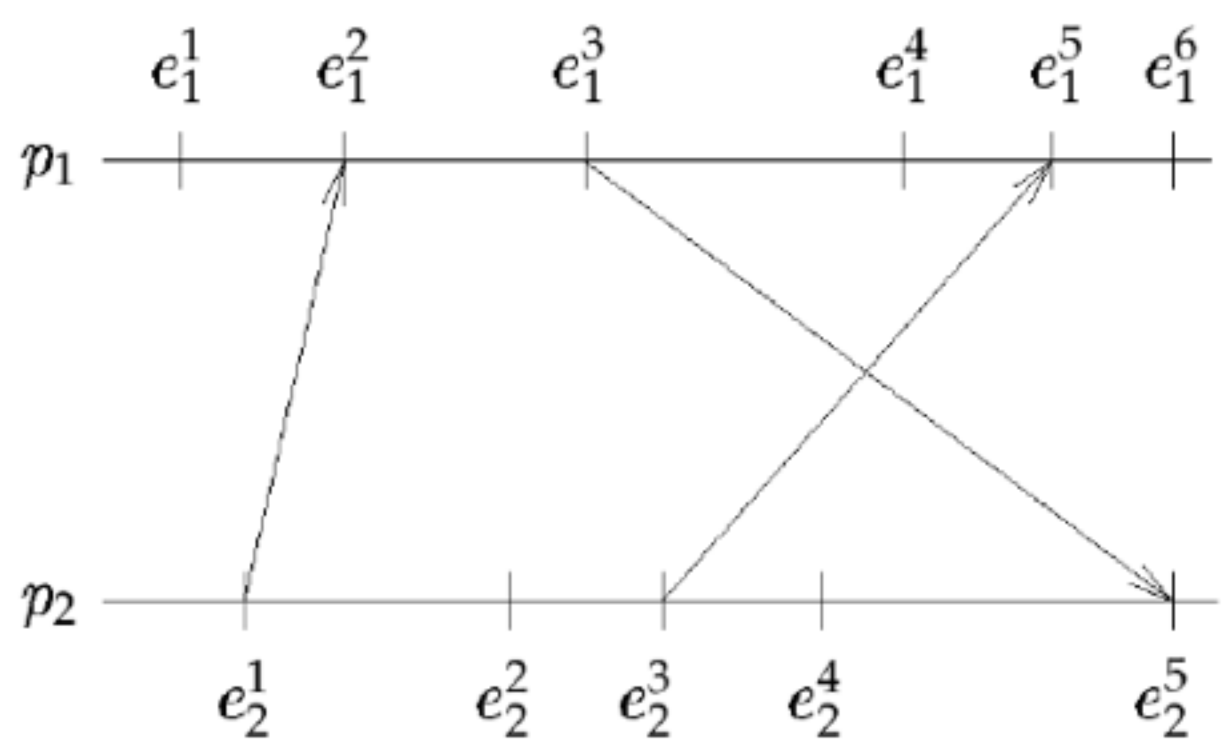

e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
p_1
p_2
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

Example

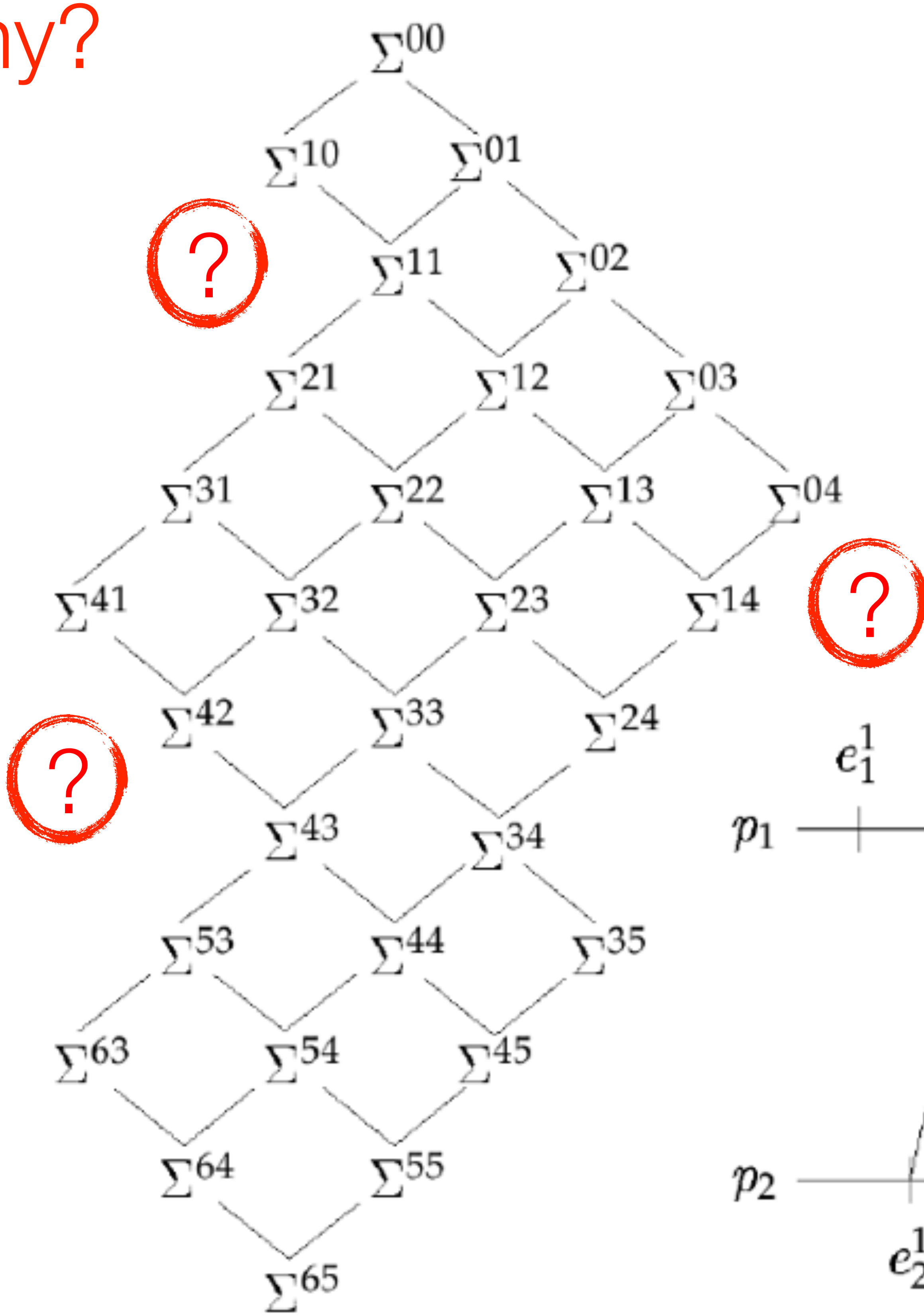


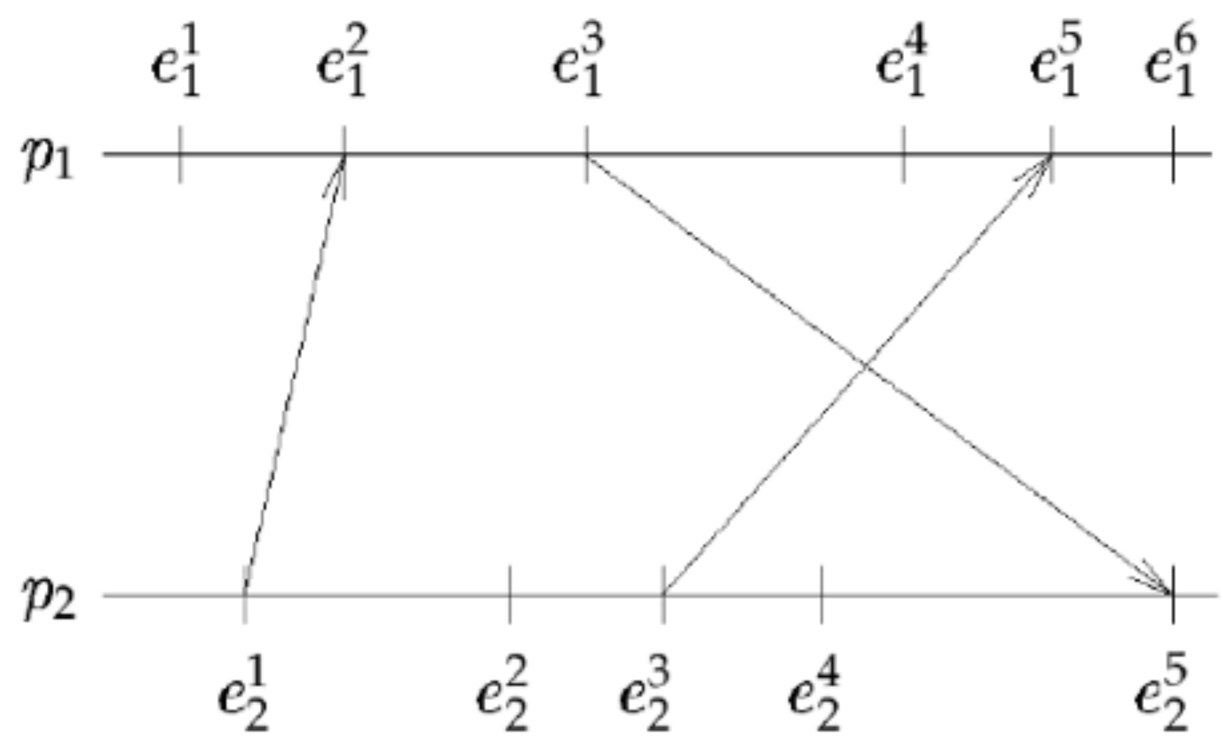
One possible **run** may pass through the sequence of global states:

$\Sigma^{00}\ \Sigma^{01}\ \Sigma^{11}\ \Sigma^{21}\ \Sigma^{31}\ \Sigma^{32}\ \Sigma^{42}\ \Sigma^{43}\ \Sigma^{44}\ \Sigma^{54}\ \Sigma^{64}\ \Sigma^{65}$



Why?


Σ^{00}
Σ^{10} Σ^{01}
Σ^{11} Σ^{02}
Σ^{21} Σ^{12} Σ^{03}
Σ^{31} Σ^{22} Σ^{13} Σ^{04}
Σ^{41} Σ^{32} Σ^{23} Σ^{14}
Σ^{42} Σ^{33} Σ^{24}
Σ^{43} Σ^{34}
Σ^{53} Σ^{44} Σ^{35}
Σ^{63} Σ^{54} Σ^{45}
Σ^{64} Σ^{55}
Σ^{65}
?
?
?


e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
p_1
p_2
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

Stable Predicates

- Consider a global state construction protocol:
  - Let Σ^a be the global state in which the protocol is **initiated**
  - Let Σ^f be the global state in which the protocol **terminates**
  - Let Σ^s be the global state **constructed** by the protocol
- Since $\Sigma^a \rightsquigarrow \Sigma^s \rightsquigarrow \Sigma^f$, if Φ is stable, then:

$$\Phi(\Sigma^s) = \textbf{true} \Rightarrow \Phi(\Sigma^f) = \textbf{true}$$

$$\Phi(\Sigma^s) = \textbf{false} \Rightarrow \Phi(\Sigma^a) = \textbf{false}$$

Non Stable Predicates

Problems of Non-Stable Predicates

- The condition encoded by the predicate may not persist long enough for it to be true when the predicate is evaluated
- If a predicate **Φ** is found to be true, we do not know whether **Φ** *ever* held during the *actual* run

Conclusions

- Evaluating a non-stable predicate *over a single computation* makes no sense
- The evaluation must be extended to the entire lattice of the computation
- It is possible to evaluate a predicate over an entire computation using an observation obtained by a passive monitor

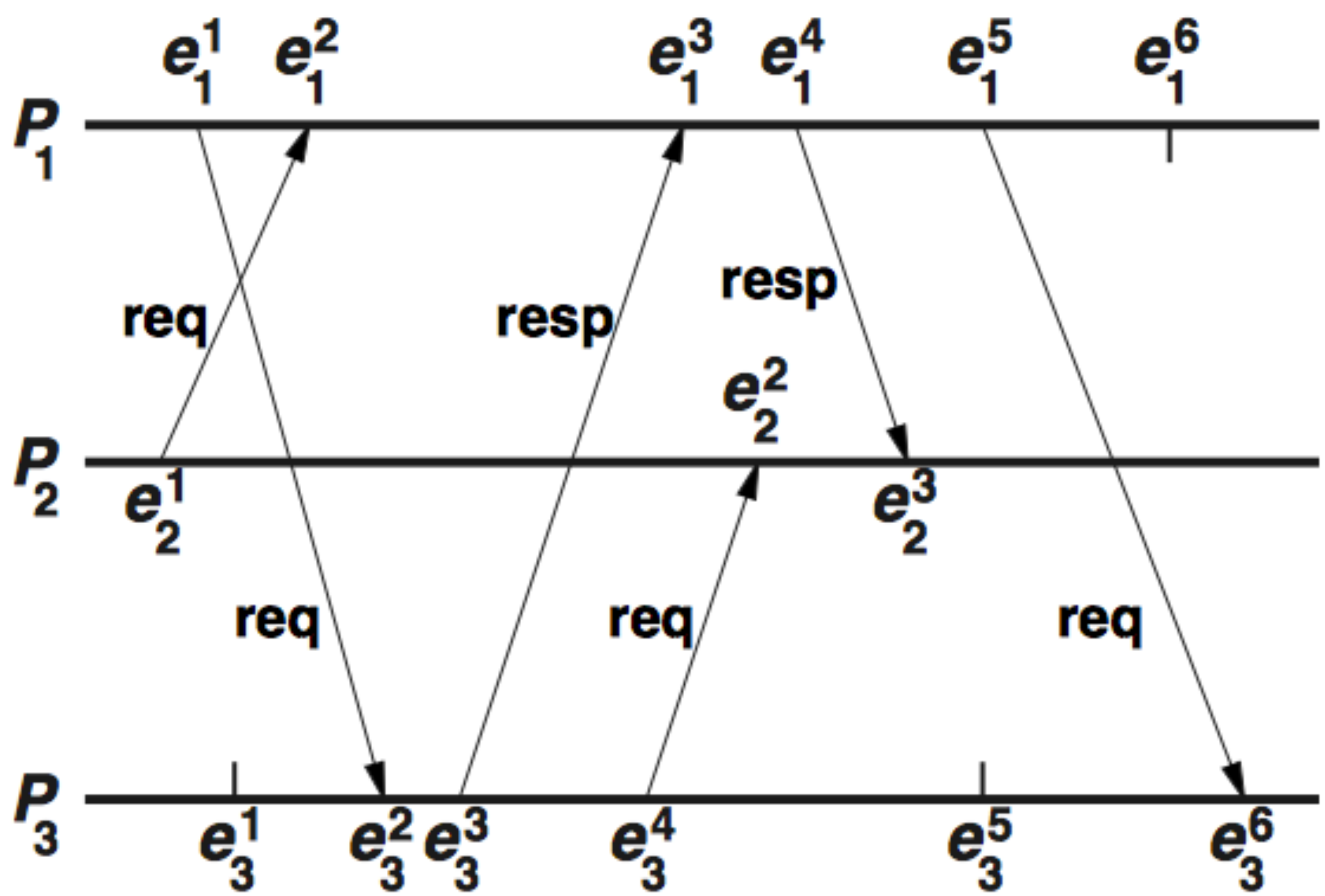
Passive Monitor

- A single process $\mathbf{p_0}$ called monitor is responsible for evaluating $\mathbf{\Phi}$
- We assume that $\mathbf{p_0}$ is distinct from $\mathbf{p_1} \dots \mathbf{p_n}$
- At each (relevant - state change) event, a node sends a message to the monitor describing it local state
- The monitor collects messages to reconstruct the global state
- The *sequence of events corresponding to the order in which notification messages arrive at the monitor* is called an **observation**
- Given the *asynchronous* nature of our distributed system, *any permutation of a run **R** is a possible observation of it*

Example of Observations

$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$

$$\begin{aligned} O_1 &= e_2^1\, e_1^1\, e_3^1\, e_3^2\, e_3^4\, e_1^2\, e_2^2\, e_3^3\, e_1^3\, e_1^4\, e_3^5 \ldots \\ O_2 &= e_1^1\, e_3^1\, e_2^1\, e_3^2\, e_1^2\, e_3^3\, e_3^4\, e_1^3\, e_2^2\, e_3^5\, e_3^6 \ldots \\ O_3 &= e_3^1\, e_2^1\, e_1^1\, e_1^2\, e_3^2\, e_3^3\, e_1^3\, e_3^4\, e_1^4\, e_2^2\, e_1^5 \ldots \end{aligned}$$



Observations vs Runs

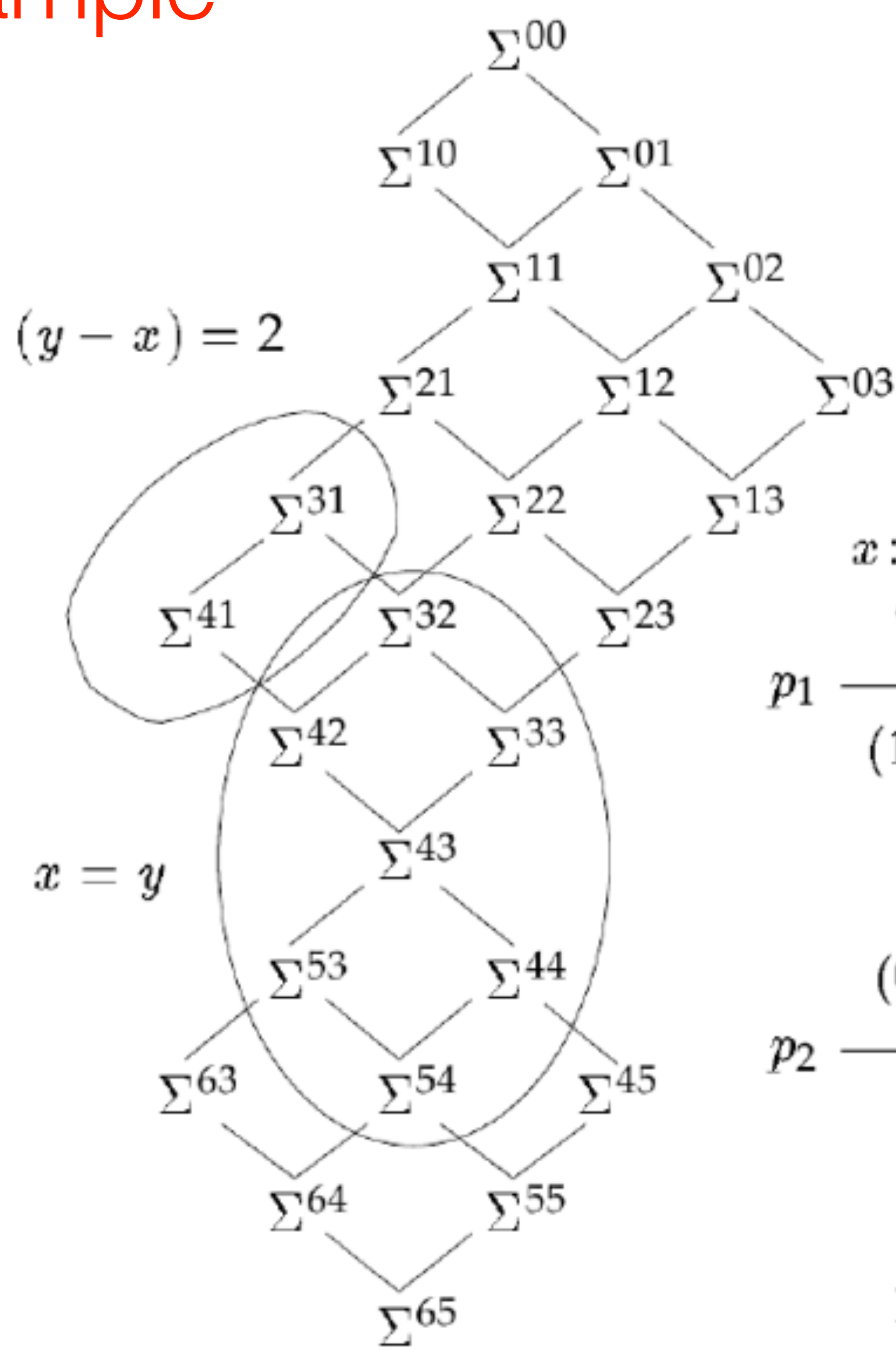
- A **run** of a distributed computation is a **total ordering R of its events** that corresponds to an **actual execution**
- An **observation** is a **total ordering Ω of events** constructed from within the system
- A **single run may have many observations**
- An **observation** can correspond to:
  - A consistent run
  - A run which is **not** consistent
  - No run at all

  Homework: can you find example of the three cases? Can you explain why this happen?
- **Consistent Observation**: An observation is **consistent** if it corresponds to a consistent run

Possibly And Definitely

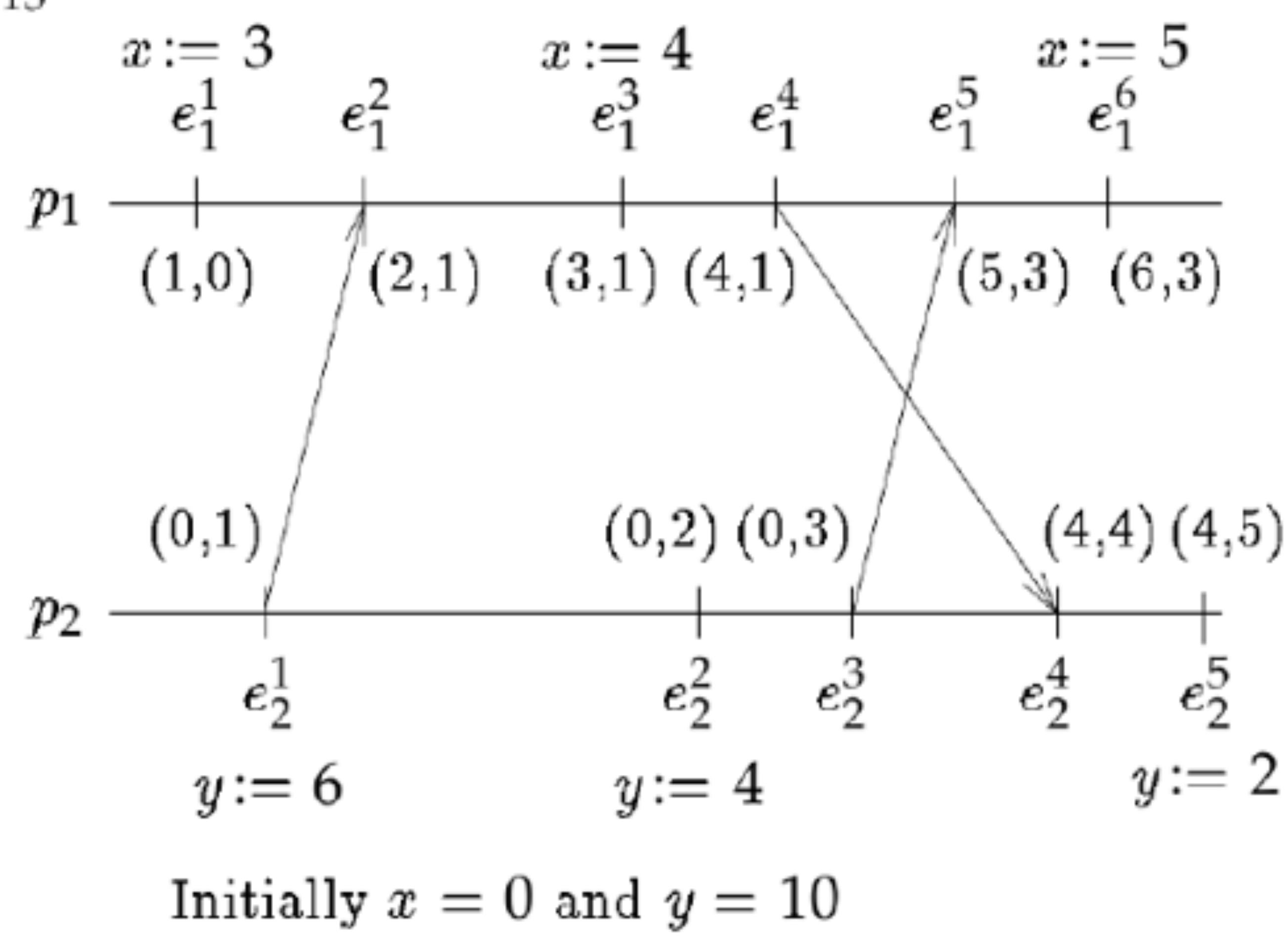
- **PROBLEM:** By means of a passive monitor, we want to know if a non-stable predicate ***possibly*** occurred or ***definitely*** occurred
  - **Possibly(Φ)**: ***There exists*** a *consistent* observation O of the computation such that **Φ** holds in a global state of O
  - **Definitely(Φ)**: ***For every*** *consistent* observation O of the computation, there exists a global state of O in which **Φ** holds
- **Debugging:** If Possibly(**Φ**) is true, and it identifies some erroneous state of the computation, than there is a bug, even if it is not observed during an actual run

Example

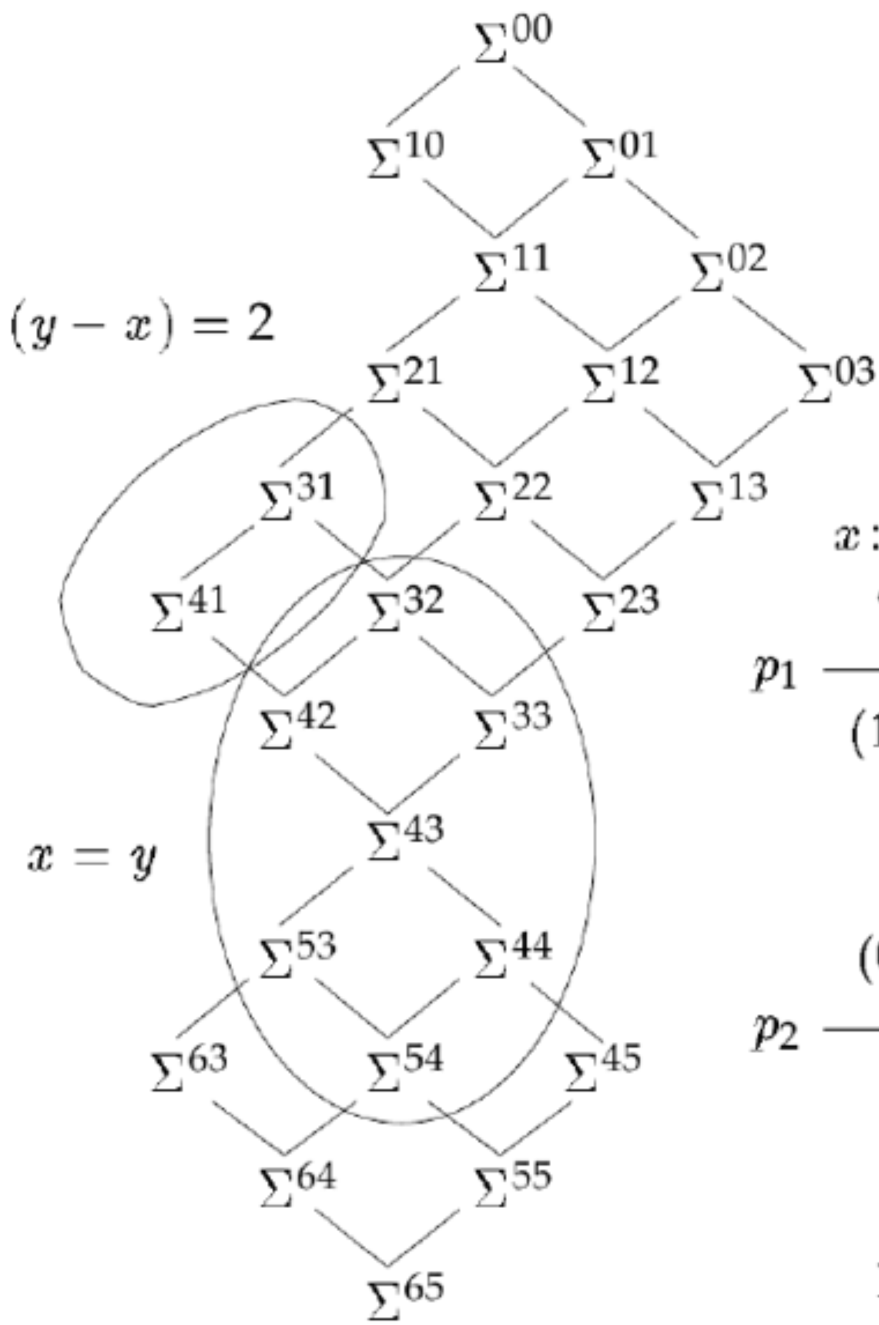


Possibly((y - x) = 2)

Definitely(x = y)

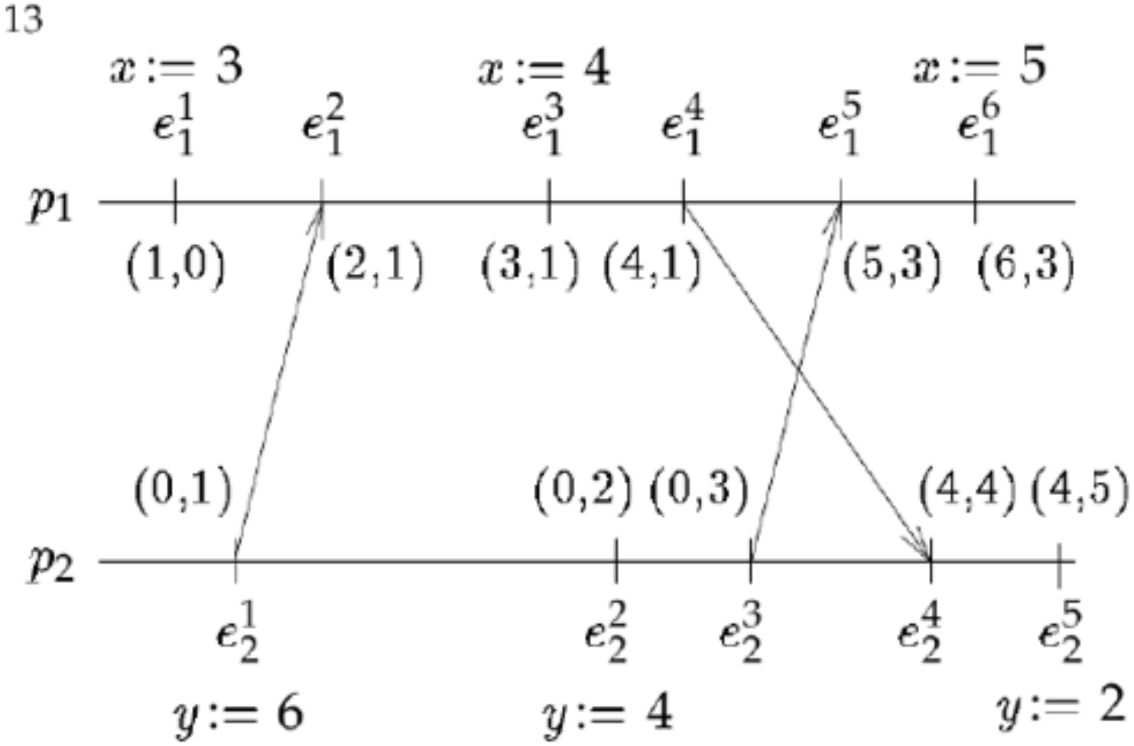


Possibly And Definitely Are Not Duals



$$\neg\textbf{Possibly}(\Phi) \not\Leftrightarrow \textbf{Definitely}(\neg\Phi)$$

$$\neg\textbf{Definitely}(\Phi) \not\Leftrightarrow \textbf{Possibly}(\neg\Phi)$$



Initially $x = 0$ and $y = 10$

Example
(of the latter):
Possibly(x ≠ y)
Definitely(x = y)

Algorithms For Detecting Possibly And Definitely

- We use the passive approach in which processes send notifications of events relevant to **Φ** to the monitor p_0
- Events are tagged with vector clocks
- The monitor collects all the events and builds the lattice of global states
- **HOMEWORK:** HOW?

Algorithm For Detecting **Possibly**

The algorithm constructs the set of global states current with progressively increasing levels (denoted by l).

When a member of current satisfies Φ, then the procedure terminates indicating that **Possibly(Φ)** holds.

If, however, the procedure constructs the final global state and finds that this global state does not satisfy Φ, then the procedure returns **¬Possibly(Φ)**.

```
procedure Possibly(Φ);
    var current: set of global states;
        ℓ: integer;
    begin
        % Synchronize processes and distribute Φ
        send Φ to all processes;
        current := global state Σ^{0...0};
        release processes;
        ℓ := 0;
        % Invariant: current contains all states of level ℓ that are reachable from Σ^{0...0}
        while (no state in current satisfies Φ) do
            if current = final global state then return false
            ℓ := ℓ + 1;
            current := states of level ℓ
        od
        return true
    end
```

Algorithm For Detecting **Definitely**

```
procedure Definitely(Φ);
    var current, last: set of global states;
        ℓ: integer;
    begin
        % Synchronize processes and distribute Φ
        send Φ to all processes;
        last := global state Σ^{0...0};
        release processes;
        remove all states in last that satisfy Φ;
        ℓ := 1;
        % Invariant: last contains all states of level ℓ − 1 that are reachable
        % from Σ^{0...0} without passing through a state satisfying Φ
        while (last ≠ { }) do
            current := states of level ℓ reachable from a state in last;
            remove all states in current that satisfy Φ;
            if current = final global state then return false
            ℓ := ℓ + 1;
            last := current
        od
        return true
    end ;
```

The algorithm iteratively constructs the set of global states (current) that have a level l and are reachable from the initial global state without passing through a global state that satisfies Φ.

If this set is empty, then **Definitely**(Φ) holds.

If this set contains only the final global state then ¬**Definitely**(Φ) holds.