Class 7
Today’s plan

- Asynchronous systems
- Formal model
  - I/O automata
  - Executions and traces
  - Operations: composition, hiding
  - Properties and proof methods:
    - Invariants
    - Simulation relations

- Reading: Chapter 8

- Next:
  - Asynchronous network algorithms: Leader election, breadth-first search, shortest paths, spanning trees.
  - Reading: Chapters 14 and 15
Last time

• Finished synchronous network algorithms:
  – Lower bounds on number of rounds
  – k-agreement
• Commit:
  – 2-phase commit:
    • Weak termination only.
  – 3-phase commit:
    • Strong termination.
    • But depends strongly on synchrony:
      – Coordinator deduces that all processes are ready or failed, just by waiting sufficiently long so it knows that its messages have arrived.
Practical issues for 3-phase commit

• Depends on strong assumptions, which may be hard to guarantee in practice:
  – Synchronous model:
    • Could emulate with approximately-synchronized clocks, timeouts.
  – Reliable message delivery:
    • Could emulate with acks and retransmissions.
    • But if retransmissions add too much delay, then we can’t emulate the synchronous model accurately.
    • Leads to unbounded delays, asynchronous model.
  – Accurate diagnosis of process failures:
    • Get this “for free” in the synchronous model.
    • E.g., 3-phase commit algorithm lets process that doesn’t hear from another process i at a round conclude that i must have failed.
    • Very hard to guarantee in practice: In Internet, or even a LAN, how to reliably distinguish failure of a process from lost communication?

• Other consensus algorithms can be used for commit, including some that don’t depend on such strong timing and reliability assumptions.
Paxos consensus algorithm

• A more robust consensus algorithm, could be used for commit.
• Tolerates process stopping and recovery, message losses and delays,…
• Runs in partially synchronous model.
• Based on earlier algorithm [Dwork, Lynch, Stockmeyer].
• Algorithm idea:
  – Processes use unreliable leader election subalgorithm to choose coordinator, who tries to achieve consensus.
  – Coordinator decides based on active support from majority of processes.
  – Does not assume anything based on not receiving a message.
  – Difficulties arise when multiple coordinators are active---must ensure consistency.
• Practical difficulties with fault-tolerance in the synchronous model motivate studying the asynchronous model.
Asynchronous systems

- No timing assumptions
  - No rounds
- Two kinds of asynchronous models:
  - Asynchronous networks
    - Processes communicating via channels
  - Asynchronous shared-memory systems
    - Processes communicating via shared objects
Asynchronous network: Processes and channels

**Q:** Mathematically speaking, what are these ps and Cs?

**A:** “Reactive” components, which interact with their environments via input and output actions.
Asynchronous shared-memory system: Processes and objects

These processes and objects are also “reactive” components.
In both cases, reactive components.
So, we give a general model for reactive components.
Specifying problems and systems

- Processes, channels, and objects are automata
  - Take actions while changing state.
  - Reactive
    - Interact with environment via input and output actions.
    - Not just functions from input values to output values, but more flexible interactions.

- Execution:
  - Sequence of actions
  - Interleaving semantics

- External behavior (trace):
  - We observe external actions.
  - State and internal actions are hidden.
  - Problems specify allowable traces.
I/O Automata
Input/Output Automata

- General **mathematical modeling framework** for reactive components.
  - Little structure---must add structure to specialize it for networks, shared-memory systems,…

- **Designed for describing systems in a modular way:**
  - Supports description of individual system components, and how they compose to yield a larger system.
  - Supports description of systems at different **levels of abstraction**, e.g.:
    - Detailed implementation vs. more abstract algorithm description.
    - Optimized algorithm vs. simpler, unoptimized version.

- **Supports standard proof techniques:**
  - Invariants
  - **Simulation relations** (like running 2 algorithms side-by-side and relating their behavior step-by-step).
  - **Compositional reasoning** (prove properties of individual components; use to infer properties for overall system).
Input/output automaton

- State transition system
  - Transitions labeled by actions
- Actions classified as input, output, internal
  - Input, output are external.
  - Output, internal are locally controlled.
Input/output automaton

- $\text{sig} = (\text{in, out, int})$
  - input, output, internal actions (disjoint)
  - $\text{acts} = \text{in} \cup \text{out} \cup \text{int}$
  - $\text{ext} = \text{in} \cup \text{out}$
  - $\text{local} = \text{out} \cup \text{int}$
- $\text{states}$: Not necessarily finite
- $\text{start} \subseteq \text{states}$
- $\text{trans} \subseteq \text{states} \times \text{acts} \times \text{states}$
  - Input-enabled: Any input “enabled” in any state.
- $\text{tasks}$, partition of locally controlled actions
  - Used for liveness.
Remarks

- A step of an automaton is an element of trans.
- Action $\pi$ is enabled in a state $s$ if there is a step $(s, \pi, s')$ for some $s'$.
- I/O automata must be input-enabled.
  - Every input action is enabled in every state.
  - Captures idea that an automaton cannot control inputs.
    - If we want restrictions, model the environment as another automaton and express restrictions in terms of the environment.
    - Could allow a component to detect bad inputs and halt, or exhibit unconstrained behavior for bad inputs.
- Tasks correspond to “threads of control”.
  - Used to define fairness (give turns to all tasks).
  - Needed to guarantee liveness properties (e.g., the system keeps making progress, or eventually terminates).
Channel automaton

- Reliable unidirectional FIFO channel between two processes.
  - Fix message alphabet M.

- signature
  - input actions: send(m), m ∈ M
  - output actions: receive(m), m ∈ M
  - no internal actions

- states
  - queue: FIFO queue of M, initially empty
Channel automaton

- **trans**
  - send(m)
    - effect: add m to (end of) **queue**
  - receive(m)
    - precondition: m is at head of **queue**
    - effect: remove head of **queue**

- **tasks**
  - All receive actions in one task.
Channel automaton

- **trans**
  - send\((m)_{i,j}\)
    - effect: add \(m\) to (end of) queue
  - receive\((m)_{i,j}\)
    - precondition: \(m\) is at head of queue
    - effect: remove head of queue

- **tasks**
  - All receive actions in one task
A process

• E.g., in a consensus protocol.
• See book, p. 205, for code details.
• Inputs arrive from the outside.
• Process sends/receives values, collects vector of values for all processes.
• When vector is filled, outputs a decision obtained as a function of the vector.
• Can get new inputs, change values, send and output repeatedly.
• Tasks for:
  – Sending to each individual neighbor.
  – Outputting decisions.
Executions

- An I/O automaton executes as follows:
  - Start at some start state.
  - Repeatedly take step from current state to new state.
- Formally, an execution is a finite or infinite sequence:
  - $s_0 \pi_1 s_1 \pi_2 s_2 \pi_3 s_3 \pi_4 s_4 \pi_5 s_5 ...$ (if finite, ends in state)
  - $s_0$ is a start state
  - $(s_i, \pi_{i+1}, s_{i+1})$ is a step (i.e., in trans)

$\lambda, \text{send}(a), a, \text{send}(b), ab, \text{receive}(a), b, \text{receive}(b), \lambda$
Execution fragments

- An I/O automaton executes as follows:
  - Start at some start state.
  - Repeatedly take step from current state to new state.
- Formally, an execution is a sequence:
  - \( s_0 \pi_1 s_1 \pi_2 s_2 \pi_3 s_3 \pi_4 s_4 \pi_5 s_5 \ldots \)
  - \( s_0 \) is a start state
  - \((s_i, \pi_{i+1}, s_{i+1})\) is a step.
Invariants and reachable states

- A state is **reachable** if it appears in some execution.
  - Equivalently, at the end of some finite execution.

- An **invariant** is a predicate that is true for every reachable state.
  - Most important tool for proving properties of concurrent/distributed algorithms.
  - Typically proved by induction on length of execution.
Traces

- Allow us to focus on components’ external behavior.
- Useful for defining correctness.
- A trace of an execution is the subsequence of external actions in the execution.
  - No states, no internal actions.
  - Denoted trace(\(\alpha\)), where \(\alpha\) is an execution.
  - Models “observable behavior”.

\[
\lambda, \text{send}(a), a, \text{send}(b), ab, \text{receive}(a), b, \text{receive}(b), \lambda
\]

\[
\text{send}(a), \text{send}(b), \text{receive}(a), \text{receive}(b)
\]
Operations on I/O Automata
Operations on I/O automata

- To describe how systems are built out of components, the model has operations for composition, hiding, renaming.

- Composition:
  - “Put multiple automata together.”
  - Output actions of one may be input actions of others.
  - All components having an action perform steps involving that action at the same time (“synchronize on actions”).

- Composing finitely many or countably infinitely many automata $A_i, i \in I$:

- Need compatibility conditions:
  - Internal actions aren’t shared:
    - $\text{int}(A_i) \cap \text{acts}(A_j) = \emptyset$
  - Only one automaton controls each output:
    - $\text{out}(A_i) \cap \text{out}(A_j) = \emptyset$
  - But output of one automaton can be an input of one or more others.
  - No action is shared by infinitely many $A_i$s.
Operations on I/O automata
Composition of compatible automata

- Compose two automata A and B (see book for general case).
  - $\text{out}(A \times B) = \text{out}(A) \cup \text{out}(B)$
  - $\text{int}(A \times B) = \text{int}(A) \cup \text{int}(B)$
  - $\text{in}(A \times B) = \text{in}(A) \cup \text{in}(B) - (\text{out}(A) \cup \text{out}(B))$
  - $\text{states}(A \times B) = \text{states}(A) \times \text{states}(B)$
  - $\text{start}(A \times B) = \text{start}(A) \times \text{start}(B)$
  - $\text{trans}(A \times B)$: includes $(s, \pi, s')$ iff
    - $(s_A, \pi, s'_A) \in \text{trans}(A)$ if $\pi \in \text{acts}(A); s_A = s'_A$ otherwise.
    - $(s_B, \pi, s'_B) \in \text{trans}(B)$ if $\pi \in \text{acts}(B); s_B = s'_B$ otherwise.
  - $\text{tasks}(A \times B) = \text{tasks}(A) \cup \text{tasks}(B)$

- Notation: $\prod_{i \in I} A_i$, for composition of $A_i : i \in I$ (I countable)
Composition of channels and consensus processes

\[
\begin{align*}
\text{Composition of channels and consensus processes} \\
p_1 & \quad C_{1,2} \quad C_{2,1} \\
\text{init}(v)_1 & \quad \text{send}(m)_{1,2} \quad \text{receive}(m)_{1,2} \\
\text{decide}(v)_1 & \quad \text{receive}(m)_{2,1} \quad \text{send}(m)_{2,1}
\end{align*}
\]
Composition: Basic results

- Projection
  - Execution of composition “looks good” to each component.

- Pasting
  - If execution “looks good” to each component, it is good overall.

- Substitutivity
  - Can replace a component with one that implements it.
Composition: Basic results

Theorem 1: Projection

- If $\alpha \in \text{execs}(\prod A_i)$ then $\alpha|A_i \in \text{execs}(A_i)$ for every $i$.
- If $\beta \in \text{traces}(\prod A_i)$ then $\beta|A_i \in \text{traces}(A_i)$ for every $i$. 
Composition: Basic results

Theorem 2: Pasting

Suppose $\beta$ is a sequence of external actions of $\prod A_i$.

- If $\alpha_i \in \text{execs}(A_i)$ and $\beta|A_i = \text{trace}(\alpha_i)$ for every $i$, then there is an execution $\alpha$ of $\prod A_i$ such that $\beta = \text{trace}(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i$.
- If $\beta |A_i \in \text{traces}(A_i)$ for every $i$ then $\beta \in \text{traces}(\prod A_i)$. 
Composition: Basic results

Theorem 3: Substitutivity

- Suppose $A_i$ and $A'_i$ have the same external signature, and $\text{traces}(A_i) \subseteq \text{traces}(A'_i)$ for every $i$.
  - A kind of “implementation” relationship.
- Then $\text{traces}(\prod A_i) \subseteq \text{traces}(\prod A'_i)$ (assuming compatibility).

Proof:

- Follows from trace pasting and projection, Theorems 1 and 2.
Other operations on I/O automata

• Hiding
  – Make some output actions internal.
  – Hides internal communication among components of a system.

• Renaming
  – Change names of some actions.
  – Action names are important for specifying component interactions.
  – E.g., define a “generic” automaton, then rename actions to define many instances to use in a system.
    • As we did with channel automata.
Fairness
Fairness

- Task T (set of actions) corresponds to a “thread of control”.
- Used to define “fair” executions: a task that is continuously enabled gets to take a step.
- Needed to prove liveness properties, e.g., that something eventually happens, like an algorithm terminating.

- Formally, execution (or fragment) \( \alpha \) of A is fair to task T if one of the following holds:
  - \( \alpha \) is finite and T is not enabled in the final state of \( \alpha \).
  - \( \alpha \) is infinite and contains infinitely many events in T.
  - \( \alpha \) is infinite and contains infinitely many states in which T is not enabled.

- Execution of A is fair if it is fair to all tasks of A.
- Trace of A is fair if it is the trace of a fair execution of A.
Example

- **Channel**
  - Only one task (all receive actions).
  - A finite execution of Channel is fair iff queue is empty at the end.
  - **Q:** Is every infinite execution of Channel fair?

- **Consensus process**
  - Separate tasks for sending to each other process, and for output.
  - Means it “keeps trying” to do these forever.
Fairness and composition

- Fairness “behaves nicely” with respect to composition---results analogous to non-fair results:

**Theorem 4: Projection**
- If $\alpha \in \text{fairexecs}(\prod A_i)$ then $\alpha|A_i \in \text{fairexecs}(A_i)$ for every $i$.
- If $\beta \in \text{fairtraces}(\prod A_i)$ then $\beta|A_i \in \text{fairtraces}(A_i)$ for every $i$.

**Theorem 5: Pasting**
Suppose $\beta$ is a sequence of external actions of $\prod A_i$.
- If $\alpha_i \in \text{fairexecs}(A_i)$ and $\beta|A_i = \text{trace}(\alpha_i)$ for every $i$, then there is a fair execution $\alpha$ of $\prod A_i$ such that $\beta = \text{trace}(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i$.
- If $\beta|A_i \in \text{fairtraces}(A_i)$ for every $i$ then $\beta \in \text{fairtraces}(\prod A_i)$.
Theorem 6: Substitutivity

- Suppose $A_i$ and $A'_i$ have the same external signature, and $\text{fairtraces}(A_i) \subseteq \text{fairtraces}(A'_i)$ for every $i$.
  - Another kind of “implementation” relationship.
- Then $\text{fairtraces}(\prod A_i) \subseteq \text{fairtraces}(\prod A'_i)$. 
Composition of channels and consensus processes

In fair executions:
- After init, keep sending latest val forever.
- All messages that are sent are delivered.
- After vector is full, output latest decision forever.
Properties and Proof Methods

- Compositional reasoning
- Invariants
- Trace properties
- Simulation relations
Compositional reasoning

- Use Theorems 1-6 to infer properties of a system from properties of its components.
- And vice versa.
Invariants

- A state is **reachable** if it appears in some execution (or, at the end of some finite execution).
- An **invariant** is a predicate that is true for every reachable state.
- Most important tool for proving properties of concurrent and distributed algorithms.
- Proving invariants:
  - Typically, by induction on length of execution.
  - Often prove batches of inter-dependent invariants together.
  - Step granularity is finer than round granularity, so proofs are harder and more detailed than those for synchronous algorithms.
Trace properties

- A trace property is essentially a set of allowable external behavior sequences.

- A **trace property** $P$ is a pair of:
  - $\text{sig}(P)$: External signature (no internal actions).
  - $\text{traces}(P)$: Set of sequences of actions in $\text{sig}(P)$.

- Automaton $A$ **satisfies** trace property $P$ if (two different notions):
  - $\text{extsig}(A) = \text{sig}(P)$ and $\text{traces}(A) \subseteq \text{traces}(P)$
  - $\text{extsig}(A) = \text{sig}(P)$ and $\text{fairtraces}(A) \subseteq \text{traces}(P)$
Safety and liveness

• **Safety property:** “Bad” thing doesn't happen:
  – Nonempty (null trace is always safe).
  – Prefix-closed: Every prefix of a safe trace is safe.
  – Limit-closed: Limit of sequence of safe traces is safe.

• **Liveness property:** “Good” thing happens eventually:
  – Every finite sequence over acts(P) can be extended to a sequence in traces(P).
  – “It's never too late.”

• Can define safety/liveness for executions similarly.
• Fairness can be expressed as a liveness property for executions.
Automata as specifications

- Every I/O automaton specifies a trace property \((\text{extsig}(A), \text{traces}(A))\).
- So we can use an automaton as a problem specification.
- Automaton A “implements” automaton B if
  - \(\text{extsig}(A) = \text{extsig}(B)\)
  - \(\text{traces}(A) \subseteq \text{traces}(B)\)
Hierarchical proofs

- Important strategy for proving correctness of complex asynchronous distributed algorithms.
- Define a series of automata, each implementing the previous one ("successive refinement").
- Highest-level automaton model captures the "real" problem specification.
- Next level is a high-level algorithm description.
- Successive levels represent more and more detailed versions of the algorithm.
- Lowest level is the full algorithm description.
Hierarchical proofs

- For example:
  - High levels centralized, lower levels distributed.
  - High levels inefficient but simple, lower levels optimized and more complex.
  - High levels with large granularity steps, lower levels with finer granularity steps.
- In all these cases, lower levels are harder to understand and reason about.
- So instead of reasoning about them directly, relate them to higher-level descriptions.
- Method similar to what we saw for synchronous algorithms.
Hierarchical proofs

- Recall, for synchronous algorithms:
  - Optimized algorithm runs side-by-side with unoptimized version, and “invariant” proved to relate the states of the two algorithms.
  - Prove using induction.

- For asynchronous systems, things become harder:
  - Asynchronous model has more nondeterminism (in choice of new state, in order of steps).
  - So, harder to determine which execs to compare.

- One-way implementation relationship is enough:
  - For each execution of the lower-level algorithm, there is a corresponding execution of the higher-level algorithm.
  - “Everything the algorithm does is allowed by the spec.”
  - Don’t need the other direction: doesn’t matter if the algorithm does everything that is allowed.
Simulation relations

- Most common method of proving that one automaton implements another.
- Assume A and B have the same extsig, and R is a relation from states(A) to states(B).
- Then R is a simulation relation from A to B provided:
  - \( s_A \in \text{start}(A) \) implies there exists \( s_B \in \text{start}(B) \) such that \( s_A R s_B \).
  - If \( s_A, s_B \) are reachable states of A and B, \( s_A R s_B \) and \( (s_A, \pi, s'_A) \) is a step, then there is an execution fragment \( \beta \) starting with \( s_B \) and ending with \( s'_B \) such that \( s'_A R s'_B \) and \( \text{trace}(\beta) = \text{trace}(\pi) \).
Simulation relations

- $R$ is a simulation relation from $A$ to $B$ provided:
  - $s_A \in \text{start}(A)$ implies $\exists s_B \in \text{start}(B)$ such that $s_A \mathrel{R} s_B$.
  - If $s_A, s_B$ are reachable states of $A$ and $B$, $s_A \mathrel{R} s_B$ and $(s_A, \pi, s'_A)$ is a step, then $\exists \beta$ starting with $s_B$ and ending with $s'_B$ such that $s'_A \mathrel{R} s'_B$ and $\text{trace}(\beta) = \text{trace}(\pi)$. 
Simulation relations

- **Theorem:** If there is a simulation relation from A to B then traces(A) \(\subseteq\) traces(B).
- This means all traces of A, not just finite traces.
- **Proof:** Fix a trace of A, arising from a (possibly infinite) execution of A.
- Create a corresponding execution of B, using an iterative construction.

\[
\begin{align*}
S_{0,A} \xrightarrow{\Pi_1} S_{1,A} \xrightarrow{\Pi_2} S_{2,A} \xrightarrow{\Pi_3} S_{3,A} \xrightarrow{\Pi_4} S_{4,A} \xrightarrow{\Pi_5} S_{5,A}
\end{align*}
\]
Simulation relations

- **Theorem:** If there is a simulation relation from A to B then $\text{traces}(A) \subseteq \text{traces}(B)$. 
Theorem: If there is a simulation relation from A to B then traces(A) ⊆ traces(B).
Simulation relations

- **Theorem:** If there is a simulation relation from A to B then \( \text{traces}(A) \subseteq \text{traces}(B) \).
Example: Channels

- Show two channels implement one.

- Rename some actions.

- Claim that $D = \text{hide}_{\{\text{pass}(m)\}} A \times B$ implements $C$, in the sense that $\text{traces}(D) \subseteq \text{traces}(C)$. 
Recall: Channel automaton

- Reliable unidirectional FIFO channel.
- Signature
  - Input actions: send(m), m ∈ M
  - Output actions: receive(m), m ∈ M
  - No internal actions
- States
  - Queue: FIFO queue of M, initially empty
Channel automaton

- **trans**
  - send(m)
    - effect: add m to queue
  - receive(m)
    - precondition: m = head(queue)
    - effect: remove head of queue

- **tasks**
  - All receive actions in one task
Composing two channel automata

- Output of B is input of A
  - Rename receive(m) of B and send(m) of A to pass(m).
- \( D = \text{hide} \{ \text{pass}(m) | m \in M \} \) \( A \times B \) implements C
- Define simulation relation \( R \):
  - For \( s \in \text{states}(D) \) and \( u \in \text{states}(C) \), \( s R u \) iff \( u.\text{queue} \) is the concatenation of \( s.A.\text{queue} \) and \( s.B.\text{queue} \)
- Proof that this is a simulation relation:
  - Start condition: All queues are empty, so start states correspond.
  - Step condition: Define “step correspondence”:
Composing two channel automata

\[ s \text{ R } u \text{ iff } u.\text{queue is concatenation of } s.\text{A.queue and } s.\text{B.queue} \]

- **Step correspondence:**
  - For each step \( (s, \pi, s') \in \text{trans}(D) \) and \( u \) such that \( s \text{ R } u \),
    define execution fragment \( \beta \) of \( C \):
      - Starts with \( u \), ends with \( u' \) such that \( s' \text{ R } u' \).
      - \( \text{trace}(\beta) = \text{trace}(\pi) \)
  - Here, actions in \( \beta \) happen to depend only on \( \pi \), and
    uniquely determine post-state.
    - Same action if external, empty sequence if internal.
Composing two channel automata

$\text{send}(m)$ \quad $\text{pass}(m)$ \quad $\text{receive}(m)$

$s \ R \ u \ \iff \ u.\text{queue} \text{ is concatenation of } s.A.\text{queue} \text{ and } s.B.\text{queue}$

- **Step correspondence:**
  - $\pi = \text{send}(m)$ in D corresponds to $\text{send}(m)$ in C
  - $\pi = \text{receive}(m)$ in D corresponds to $\text{receive}(m)$ in C
  - $\pi = \text{pass}(m)$ in D corresponds to $\lambda$ in C

- **Verify that this works:**
  - Actions of C are enabled.
  - Final states related by relation R. case analysis.

- Routine case analysis:
Showing R is a simulation relation

\[ s \ R \ u \iff u.\text{queue} \text{ is concatenation of } s.A.\text{queue} \text{ and } s.B.\text{queue} \]

- **Case:** \( \pi = \text{send}(m) \)
  - No enabling issues (input).
  - Must check \( s' \ R \ u' \).
    - Since \( s \ R \ u \), \( u.\text{queue} \text{ is the concatenation of } s.A.\text{queue} \text{ and } s.B.\text{queue} \).
    - Adding the same \( m \) to the end of \( u.\text{queue} \text{ and } s.B.\text{queue} \text{ maintains the correspondence.} \)

- **Case:** \( \pi = \text{receive}(m) \)
  - Enabling: Check that \( \text{receive}(m) \), for the same \( m \), is also enabled in \( u \).
    - We know that \( m \) is first on \( s.A.\text{queue} \).
    - Since \( s \ R \ u \), \( m \) is first on \( u.\text{queue} \).
    - So enabled in \( u \).
  - \( s' \ R \ u' \): Since \( m \) removed from both \( s.A.\text{queue} \text{ and } u.\text{queue} \).
Showing $R$ is a simulation relation

$s R u$ iff $u$.queue is concatenation of $s.A$.queue and $s.B$.queue

**Case:** $\pi = \text{pass}(m)$

- No enabling issues (since no high-level steps are involved).
- Must check $s' R u$:
  - Since $s R u$, $u$.queue is the concatenation of $s.A$.queue and $s.B$.queue.
  - Concatenation is unchanged as a result of this step, so also $u$.queue is the concatenation of $s'.A$.queue and $s'.B$.queue.
Next lecture

- Basic asynchronous network algorithms:
  - Leader election
  - Breadth-first search
  - Shortest paths
  - Spanning trees.

- Reading:
  - Chapters 14 and 15
6.852J / 18.437J Distributed Algorithms
Fall 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.