Discrete Event Systems – lecture summary

Andreas Müller
March 6, 2007

Abstract
This summary is based on the slides Discrete Event Systems by Christoph Stamm, WS06/07.

Contents

1 Automata and Languages
1.1 Definitions

Alphabet A set of symbols (characters, letters).

String A string or word is a sequence of symbols of an alphabet.
  ▶ The empty string contains no symbols and is denoted by $\varepsilon$ (don’t confuse with empty set $\emptyset$, which is not a string, but rather a set of no strings).
  ▶ The length of the string is the number of symbols, e.g. $|\varepsilon| = 0$.

Regular Language A language $L$ is called a regular language, if there is a FA, that accepts the language $L$. All finite languages are regular.

Infinite Language A language is infinite if it contains an infinite number of strings. A language can be infinite but regular (example: The language that consists of binary strings with an odd number of ones).

1.2 Regular Operations
  ▶ Union ($\cup$): Match one of the patterns
  ▶ Concatenation ($\circ$): Match patterns in sequence
  ▶ Kleene-star ($*$): Match pattern 0 or more times
  ▶ Kleene-plus ($+$): Match pattern 1 or more times

The result of a regular operator applied to a regular language is always a regular language. In other words, regular languages are closed under the operations of union, concatenation and Kleene-star/plus.
1.3 Finite Automata (FA) [1/11]

1.3.1 Definition: Finite Automaton

A finite automaton (FA) is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\) where

- \(Q\) is a finite set called the states
- \(\Sigma\) is a finite set called the alphabet
- \(\delta : Q \times \Sigma \rightarrow Q\) is the transformation function
- \(q_0 \in Q\) is the start state
- \(F \subseteq Q\) is the set of accept states (final states)

1.3.2 Definition: Accepted Language

The language accepted (or recognized) by a finite automaton \(M\) is the set of all strings accepted by \(M\). It is denoted by \(L(M)\).

- Not all languages can be described as the accepted language of an FA.

1.4 Regular Expressions (REX) [1/22]

1.4.1 Regular Expressions in UNIX

Regular Operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>UNIX version</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>(\bigcup)</td>
<td></td>
<td>match one of the patterns</td>
</tr>
<tr>
<td>Concatenation</td>
<td>•</td>
<td>implicit</td>
<td>match patterns in sequence</td>
</tr>
<tr>
<td>Kleene-star</td>
<td>*</td>
<td>*</td>
<td>match pattern 0 or more times</td>
</tr>
<tr>
<td>Kleene-plus</td>
<td>+</td>
<td>+</td>
<td>match pattern 1 or more times</td>
</tr>
</tbody>
</table>

Match between 10 and 15 times: \(\{10, 15\}\)

Operator precedence is \(*, \cdot, \bigcup\)

Command Line:

- `egrep -i 'expression'`
  - `-i`: Ignore Case
- `perl -wlne'print if /expression/' $file`
- UNIX searches for lines containing the given string, to search for lines consisting of the string:
  - `egrep `^expression$``

1.5 Non-deterministic Finite Automata (NFA) [1/48]

An NFA is an automaton whose transitions don’t need to be deterministic, i.e. there can be multiple transitions from a state for the same symbol. If the automaton encounters such a transition, it is assumed to be in all possible states simultaneously. Any labeled graph with a start state is an NFA.
**Definition:** A non-deterministic finite automaton (NFA) is encapsulated by \( M = (Q, \Sigma, \delta, q_0, F) \) in the same way as an FA, except that \( \delta \) has different domain and co-domain: \( \delta : Q \times \sigma \rightarrow P(Q) \), where \( P(Q) \) is the power set of \( Q \), so that \( \delta(q, a) \) is the set of all endpoints of edges from \( q \) which are labeled by \( a \).

**Definition:** A string \( u \) is accepted by an NFA \( M \) iff there exists a path starting at \( q_0 \) which is labeled by \( u \) and ends in an accept state. The language accepted by \( M \) is the set of all strings accepted by \( M \) and is denoted by \( L(M) \).

### 1.5.1 Regular Operations for NFA

NFA are closed under regular operations. Regular operations can be applied easily to NFA:

- **Union** \( (A \cup B) \): Put \( A \) and \( B \) in parallel. Create new start state and connect it to all former start states with \( \varepsilon \)-edges.

- **Concatenation** \( (A \cdot B) \): Use start states from \( A \), connect all accept states of \( A \) to all start states from \( B \) via \( \varepsilon \)-edges and use accept states from \( B \) as new accept states.

- **Kleene-plus** \( (A+) \): Loop back all accept states to start states via \( \varepsilon \)-edges.

- **Kleene-star** \( (A*) \): Same as Kleene-plus, but create a new, accepting start state that links to the old start state with an \( \varepsilon \)-edge.

### 1.6 State Minimization

- omit non-reachable states
- find equivalent states and join them

### 1.7 Transitions between FA, NFA and REX

FA, NFA and REX are equivalent.

#### 1.7.1 NFA→FA

- always minimize the automaton first
- create power set of states \( (n \text{ states } \rightarrow 2^n \text{ states}) \)
- each new state that contains an accept state becomes an accept state
- new start state: set of old start state and all states that can recursively reached from there with \( \varepsilon \)

#### 1.7.2 REX→NFA [1/69]

\( \rightarrow \) slide 1/69
1.7.3 NFA→REX [1/83ff]
Via generic nondeterministic finite automaton (GNFA):
   - a GNFA is an automaton whose edges are labeled by regular expressions with
     - a unique start state with in-degree 0 (but arrows to every other state)
     - a unique accept state (not the start state) with out-degree 0 (but arrows from every other state)
     - (an arrow from any state to any other state)

Conversion algorithm
1. construct GNFA from NFA
   (a) if there is more than one arrow from one state to another, unify them using \( \bigcup \)
   (b) create an unique start state with in-degree 0
   (c) create an unique accept state with out-degree 0
   (d) (if there is no arrow from one state to another, insert one with label \( \emptyset \))
2. loop: as long as GNFA has more than 2 states, rip out arbitrary interior state and update labels
3. the last edge is labeled with the regular expression

1.8 Pumping Lemma [1/94]
For every regular language \( L \), there is a number \( p \) (pumping numer), such that any string in \( L \) of length \( \geq p \) is pumpable within its first \( p \) letters.

In other words, for all \( u \in L \) with \( |u| \geq p \), we can write:
   - \( u = xyz \) (\( x \) is a prefix, \( y \) is a suffix)
   - \( |y| \geq 1 \) (mid-portion is non-empty)
   - \( |xy| \leq p \) (pumping occurs in first \( p \) letters)
   - \( xy^iz \in L \) for all \( i \geq 0 \) (can pump \( y \)-portion)

(the pumping lemma may be able to prove that a language is non-regular, but not that a language is regular)

2 Smarter Automata
2.1 Context free grammars (CFG) [2/5]
Definition: A context free grammar consists of \((V, \Sigma, R, S)\) with
   - \( V \): a finite set of variables (or symbols, or non-terminals)
   - \( \Sigma \): a finite set of terminals (or the alphabet)
\( R: \) a finite set of rules (or productions) of the form \( v \to w \) with \( v \in V \) an \( w \in (\Sigma \epsilon \cup V)^* \)

\( S \in V: \) the start symbol

2.1.1 Derivation tree (parse tree) [2/10]

- each node is a symbol
  - root is the start symbol
  - parents are variables (non-terminals)
  - leaves are terminals
- leaves spell out the derived tree from left to right

2.1.2 Ambiguity

Definition: A string \( x \) is said to be ambiguous relative the grammar \( G \) if there are two essentially different ways to derive \( x \) in \( G \). I.e. \( x \) admits two or more different parse trees (or \( x \) admits different left-most or right-most derivations). A grammar \( G \) is ambiguous if it \( L(G) \) contains an ambiguous string.

2.1.3 Chomsky Normal Form [2/36]

We want only rules of the following forms:

- \( S \to \epsilon \) (only start state may produce \( \epsilon \))
- \( A \to BC \) (dyadic variable productions)
- \( A \to a \) (unit terminal productions)

Conversion:

1. Ensure that start variable doesn’t appear on the right hand side of any rule (if it does, exchange it by the productions of \( S \)).
2. Remove all epsilon productions except from start variable (e.g.: \( A \to \epsilon|a|b|AB \) becomes \( A \to a|b|AB|B \) etc)
3. Remove all unit variable productions of the form \( A \to B \).
4. Add new variables and dyadic (2) variable rules to replace non-dyadic or non-variable productions.

2.2 Pushdown Automata (PDA) [2/21]

Definition: A pushdown automaton is a 6-tuple \( M = (Q, \Sigma, \Gamma, \delta, q_0, F) \) with

- \( Q \) (states), \( \Sigma \) (input alphabet), \( q_0 \) (start state), \( F \) (accept states) are the same as for FA
- \( \Gamma \) is the stack alphabet
- \( \delta \), for a given state, input letter and stack letter gives an output letter and a stack replacement:

\[ \delta : Q \times \Sigma \times \Gamma \to P(Q \times \Gamma) \]
Labeling convention:

If at state $p$ with input $x$ and stack $y$, go to state $q$ and replace $y$ on stack with $z$.
- $x = \varepsilon$: ignore input, don’t read
- $y = \varepsilon$: ignore top of stack and push $z$
- $z = \varepsilon$: pop $y$

2.3 Tandem Pumping [2/46]

Tandem Pumping can prove that a language is not a context free grammer. To prove that a language is a CFG, a CFG that creates the language or an automaton (DFA, NFA or REX) that accepts it can be given.

2.4 Transducer [2/50]

- A finite state transducer (FST) is a type of finite automaton whose output is a string instead of accept or reject.
- Each transition is labeled with two symbols $a/b$ where $a$ is the input symbol (as for automata) and $b$ the output symbol.

2.5 Turing Machine [2/52]

Definition A Turing machine (TM) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ with
- $Q$ (states), $\Sigma$ (input alphabet), $q_0$ (start state) are the same as for FA
- $q_{\text{acc}}$ and $q_{\text{rej}}$ are accept and reject states
- $\Gamma$ is the tape alphabet, which necessarily contains the blank symbol $\square$ and the input alphabet $\Sigma$
- $\delta$ is the transition

\[
\delta : (Q - \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\})
\]

where $L, R$ is a left or right shift on the tape

A string $s$ is accepted if the TM with $s$ on the tape and the head on the left most state eventually enters the accept state.

Labeling convention:

If at state $p$ with band input $x$, write $y$ to band and go right on the tape and to state $q$.

If at state $p$ with band input $\square$ (empty), leave $\square$ on band and go left on the tape and to state $q$. 
3 Specification Models

3.1 State Charts [3/2]

State charts are an extension of automata:

- notation:

  \[
  \begin{array}{c}
  \text{event[condition]/reaction} \\
  \rightarrow \\
  \end{array}
  \]

  \[
  A \rightarrow B
  \]

  - event: can be either internally or externally generated; logical function of multiple events is allowed
  - condition: refers to values or variables that keep their value until they are reassigned; multiple conditions possible
  - the transition is enabled if event and condition is true
  - reaction: assignment to variables and/or creation of events

- Superstates
  - OR-superstates \( \rightarrow \) exactly one of the substates is active, when the superstate is active
  - AND-superstates \( \rightarrow \) all substates are active, when the superstate is active

3.2 Petri Nets [3/22]

**Definition:** A Petri net is a bipartite, directed graph defined by a tuple \((S, T, F, M_0)\), where

- \( S \) is a set of places \( p_i \)
- \( T \) is a set of transitions \( t_i \)
- \( F \) is a set of edges (flow relations) \( f_i \)
- \( M_0 : S \rightarrow N \) is the initial marking

A transition may fire if all incoming edges have the possibility to consume a token. All places at outgoing edges will then get a token. Which transition will fire is non-deterministic.

3.2.1 Analysis

**Properties of petri nets [3/38]:**

- **Reachability** A marking \( M_n \) is reachable iff there exists a sequence of firings \( \{t_1, t_2, ..., t_n\} \) so that \( M_n = M_0 \cdot t_1 \cdot t_2 \cdot ... \cdot t_n \)
- **\( K \)-Boundedness** A petri net is \( K \)-bounded if the number of tokens in every place never exceeds \( K \)
- **Safety** 1-boundedness: Every node holds at most 1 token at any time
- **Liveness** starting from the current state, can we eventually fire any transition?
Properties of transitions [3/39]: A transition $t$ in a Petri net $(N, M_0)$ is

- **dead** if $t$ cannot be fired in any firing sequence of $L(M_0)$
- **L1-live** if $t$ can be fired at least once in some sequence of $L(M_0)$
- **L2-live** if, $\forall k \in \mathbb{N}^+$, $t$ can be fired at least $k$ times in some sequence of $L(M_0)$
- **L3-live** if $t$ appears infinitely often in some infinite sequence of $L(M_0)$
- **L4-live** (live) if $t$ is L1-live for every marking reachable from $M_0$

Coverability Tree [3/42]: The coverability tree is a tree of all reachable token distributions. To denote an arbitrary number of tokens, a special symbol $\omega$ is used.

Algorithm to create the tree:
- label initial marking $M_0$ as root and tag it as new
- **while** new markings exist, pick one, say $M$ and do
  - if $M$ is identical to a marking on the way from the root to $M$, mark it as old; **continue**
  - if no transitions are enabled at $M$, tag it as deadend
  - for each enabled transition $t$ at $M$ do
    - obtain marking $M' = M \cdot t$
    - if there exists a marking $M''$ on the way from the root to $M$ s.t. $M'(p) \geq M''(p)$ for each place $p$ and $M' \neq M''$, replace $M'(p)$ with $\omega$ for $p$ where $M'(p) > M''(p)$
    - introduce $M'$ as a node, draw an arrow with label $t$ from $M$ to $M'$ and tag $M'$ as new

Results from the Coverability Tree $T$:
- the net is **bounded** iff $\omega$ does not appear in any node label
- the net is **safe** if only 0 and 1 appear in the node labels of $T$
- a transition $t$ is **dead** iff it does not appear as edge in $T$
- if $M$ is reachable from $M_0$, then there exists a node $M'$ s.t. $M \leq M'$ (necessary but not sufficient)
- for bounded petri nets, this tree is also called **reachability tree**; it contains all reachable markings
- **deadlocks** are possible if at least one node is labeled with deadend

Incidence Matrix: The incidence matrix $A$ describes the token-flow according for the different transitions:
- $A_{ij} = \text{gain of tokens at node } i \text{ when transition } j \text{ fires}$
- in other words, $A$ is a matrix with a row for each place and and column for each transition; $-n$ denotes an outgoing transition of capacity $n$; $+n$ denotes an incoming transition of capacity $n$
A marking $M$ is written as a column vector.

The firing vector $u_i$ describes the firing of transition $i$ and consists of all 0, except for the $i$-th position, where it has a 1.

A firing of transition $t_i$ at with a marking $M_k$ is written as

$$M_{k+1} = M_k + A \cdot u_i$$

Reachability:

- A marking $M_k$ is reachable from $M_0$ if there is a sequence of transitions $\{t_1, ..., t_k\}$ such that $M_k = M_0 \cdot t_1 \cdot ... \cdot t_k$

Written with the incidence matrix:

$$M_k = M_0 + A \sum_{i=1}^k u_i$$

This can be rewritten as

$$M_k - M_0 = \Delta M = A \cdot \bar{x}$$

If $M_k$ is reachable, this equation must have a solution where all components of $\bar{x}$ are positive integers (necessary but not sufficient)

### 3.2.2 Extensions [3/52]

- **weighted edges [3/30]**
  - Each edge $f_i$ has an associated weight $W(f_i)$
  - Transition is active if each place $p_i$ connected through incoming edge $f_i$ contains at least $W(f_i)$ tokens
  - New tokens are generated based on weight of edges
  - Still a regular petri net

- **finite capacity [3/31]**
  - Each place $p_i$ can hold $K(p_i)$ tokens at most
  - To remove this constraint for $p$, add a complementary place $p'$, in way that the number of tokens in $p$ and $p'$ together is always $K(p)$
  - Still a regular petri net

- **colored petri nets: token carry values (colors)**
  - Can be transformed to regular petri net

- **continuous petri net: token number can be real**
  - Can not be transformed to regular petri net

- **inhibitor arcs: enable transition if a place contains no tokens**
  - Can not be transformed to regular petri net

9
4 Stochastic discrete event systems

4.1 Basics [4/5]
→ see other summaries

4.2 Stochastic processes in discrete time [4/16]

4.2.1 Markov-Chain

- next state only depends on current state, not on past states
- use probability matrices \((P)\) for description:

\[
q_{t+1} = q_t \cdot P \\
q_t = q_0 \cdot P^t
\]

(sanity check: row sums have to be 1)

- transition times (erwartete Übergangszeiten)

\[
h_{ij} = 1 + \sum_{k: k \neq j} p_{ik} h_{kj}
\]

(if the expectation values \(h_{ij}\) and \(h_{kj}\) exist)

- arrival probabilities (Ankunftswahrscheinlichkeiten)

\[
f_{ij} = p_{ij} + \sum_{k: k \neq j} p_{ik} f_{kj}
\]

- stationary analysis [4/38]
  - behaviour for \(t \to \infty\)
  - a vector \(\pi\) with \(\pi_j \geq 0\) and \(\sum_{j \in S} \pi_j = 1\) is called a stationary distribution of the Markov chain with transition matrix \(P\), if \(\pi = \pi \cdot P\)
  - the stationary distribution \(\pi\) is an Eigenvector of \(P\) with Eigenvalue 1
  - "TI"
    * `eigvl(P)`: eigenvalues
    * `eigvc(PT)`: eigenvectors in columns
    * `eigvc((PT)^T)[2]`: second eigenvector as row vector
    * `ans(1)/rownorm(ans(1))`: rescale to element sum=1

4.3 Stochastic processes in continuous time [4/50]

4.3.1 Distributions

(from WahrStat summary)

- Gleichverteilung auf \([a, b]\), \(a < b\)
• Dichte
  \[ f_{a,b}(x) = \begin{cases} \frac{1}{b-a}, & x \in [a, b] \\ 0, & x \notin [a, b] \end{cases} \]

• Verteilungsfunktion
  \[ F_{a,b}(x) = \begin{cases} 0, & x \leq a \\ \frac{x-a}{b-a}, & a \leq x \leq b \\ 1, & x \geq b \end{cases} \]

- \( E[X] = \frac{a+b}{2} \)
- \( \text{Var}[X] = \frac{(b-a)^2}{12} \)

► Normalverteilung mit Parametern \( \mu \in \mathbb{R}, \sigma > 0 \):
  - Dichte:
    \[ f_{\mu,\sigma}(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad x \in \mathbb{R} \]
  - \( E[X] = \mu \)
  - \( \text{Var}[x] = \sigma^2 \)
  - Die Zufallsvariable \( Z \) mit Verteilung \( N(\mu, \sigma) \) (Normalverteilung mit Parametern \( \mu, \sigma \)) entspricht der Zufallsvariablen \( \frac{Z-\mu}{\sigma} \) mit Verteilung \( N(0, 1) \) (nützlich bei Verwendung einer Tabelle).

► Exponentialverteilung mit Parameter \( \lambda > 0 \):
  - Dichte:
    \[ f_{\lambda}(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0 \\ 0, & \text{sonst} \end{cases} \]
  - Verteilungsfunktion:
    \[ F_{\lambda}(x) = \int_{-\infty}^{x} f_{\lambda}(u) \, du = \begin{cases} 0, & x \leq 0 \\ 1 - e^{-\lambda x}, & x > 0 \end{cases} \]
  - \( E[X] = \frac{1}{\lambda} \)
  - \( \text{Var}[X] = \frac{1}{\lambda^2} \)
  - sind \( X_1, \ldots, X_n \) exponentialverteilt mit Parametern \( \lambda_1, \ldots, \lambda_n \), so ist das Minimum \( \min(X_1, \ldots, X_m) \) exponentialverteilt mit Parameter \( \lambda = \lambda_1 + \ldots + \lambda_n \)

4.3.2 Markov-chain in continuous time [4/57]

► residence probabilities (?) (Aufenthaltswahrscheinlichkeiten) [4/62]
  - start distribution \( q(0) \): \( q_i(0) = Pr[X(0) = i] \) for \( i \in S \) (S: states)
  - distribution at time \( t \): \( q_i(t) = Pr[X(t) = i] \) for \( i \in S \)
  - change of residence probabilities:
    \[ \frac{d}{dt} q_i(t) = \sum_{j \neq i} q_j(t) \cdot v_{j,i} - q_i(t) \cdot v_i \]
not easy to solve

for a stationary distribution, \( \frac{d}{dt} q_i(t) = 0 \) must hold

for \( t \to \infty \), we get a linear equation system, that must be solved by a stationary distribution \( \pi \):

\[
0 = \sum_{j \neq i} \pi_j \cdot v_{j,i} - \pi_i \cdot v_i
\]

see [4/65] for an example with two states

### 4.3.3 Kendall-notation \( X/Y/m \) for queues [4/68]

- \( X \) stands for the distribution of the time between two incoming jobs
- \( Y \) stands for the distribution of the job processing time (without waiting time)
- \( m \) is the number of servers
- distributions are:
  - \( D \): deterministic (fixed time)
  - \( M \): memoryless (exponential distribution)
  - \( G \): general (something else)

### 4.3.4 \( M/M/1 \)-queues [4/70]

- arrival time distances and processing time exponentially distributed with parameter \( \lambda \) resp. \( \mu \)
- traffic density: \( \rho = \frac{\lambda}{\mu} \)
- model with a Markov-chain in continuous time
  - states: number of jobs in system
  - state set: \( S = \mathbb{N}_0 \)
  - transit rate from \( i \) to \( i + 1 \) is \( \lambda \)
  - transit rate from \( i > 0 \) to \( i - 1 \) is \( \mu \)

stationary distribution [4/71]

- for \( \rho \geq 1 \): no stationary solution, queue grows infinitely
- for \( \rho < 1 \): stationary distribution \( \pi \) with \( \pi_k = (1 - \rho)^k \rho^k \) for \( k \geq 0 \). Average server load (Auslastung): \( 1 - \pi_0 = \rho \)
- expected number of jobs in the system (queue+server):

\[
N = \frac{\lambda}{\mu - \lambda} = \frac{\rho}{1 - \rho}
\]
- average response time:
  \[ T = \frac{N}{\lambda} = \frac{1}{\mu - \lambda} \]
- average waiting time without handling time:
  \[ W = T - \frac{1}{\mu} = \frac{\rho}{\mu - \lambda} \]
- average number of jobs in queue:
  \[ N_Q = \lambda W = \frac{\rho^2}{1 - \rho} \]

- generalisation of M/M/1-queue: see [4/87]

### 4.3.5 Little’s Law [4/75]

**Definitions:**
- \( N(t) \): number of jobs in the system (queue+server) at time \( t \)
- \( \alpha(t) \): number of jobs that arrived in interval \([0, t]\)
- \( T_i \): response time for job \( i \) (waiting+handling)

**Average values at time \( t \):**

\[
N_t = \frac{1}{t} \int_0^t N(\tau) d\tau, \quad \lambda_t = \frac{\alpha(t)}{t}, \quad T_t = \frac{\sum_{i=1}^{\alpha(t)} T_i}{\alpha(t)}
\]

**Limit for \( t \to \infty \):**

\[
N = \lim_{t \to \infty} N_t, \quad \lambda = \lim_{t \to \infty} \lambda_t, \quad T = \lim_{t \to \infty} T_t
\]

If these limits exists and \( \lim_{t \to \infty} \frac{\beta(t)}{t} \) exists and equals \( \lambda \) (\( \beta(t) \) is the number of finished jobs in interval \([0, t]\)), then Little’s Law tells us:

\[
N = \lambda \cdot T
\]

(this holds for strategies other than FCFS as well)

### 4.3.6 Time-Sharing

see [4/83]

### 4.3.7 M/M/1-queue with limited (to \( n \)) number of waiting spaces

- see [4/88]

- stationary distribution for waiting spaces:
  \[
  \pi_0 = \frac{1}{\sum_{i=0}^n \rho^i} = \begin{cases} 
  \frac{n+1}{1-\rho} & \text{for } \rho = 1 \\
  \frac{1}{1-\rho^n} & \text{otherwise}
  \end{cases}
  \]
  \[
  \pi_k = \rho^k \cdot \pi_0 \quad \text{for } 1 \leq k \leq n
  \]
4.3.8 $M/M/m$-System

- see [4/90]
- equilibrium if $\rho = \frac{\lambda}{m\mu} < 1$
- stationary distribution for waiting spaces:

$$\pi_0 = \frac{1}{\sum_{k=0}^{m-1} \frac{m}{k!} \frac{(\rho m)^k}{m(1-\rho)}}$$

$$\pi_k = \begin{cases} \pi_0 \cdot \frac{\lambda^k}{m^k \mu^k} & \text{for } 1 \leq k \leq m \\ \pi_0 \cdot \frac{\rho^k m^m}{m!} & \text{for } k \geq m \end{cases}$$

- probability $P_Q$ that an incoming job has to wait

$$P_Q = \frac{(\rho m)^m / (m!(1-p))}{\sum_{k=0}^{m-1} \frac{m}{k!} \frac{(\rho m)^k}{m(1-\rho)}} \quad \text{(for } \rho = \frac{\lambda}{m\mu} < 1)$$

- expectation value for number of jobs in queue

$$N_Q = P_Q \cdot \frac{\rho}{1-\rho}$$

- average wait time

$$W = \frac{N_Q}{\lambda} = P_Q \cdot \frac{\rho}{\lambda(1-\rho)}$$

- average response time

$$T = W + \frac{1}{\mu} = \frac{P_Q}{m\mu - \lambda} + \frac{1}{\mu}$$

- average number of jobs in the system

$$N = \lambda T = \frac{\rho P_Q}{1-\rho} + m\rho$$

4.3.9 $M/M/m/m$-System

- see [4/94]
- $m$ servers, at most $m$ jobs
- stationary distribution

$$\pi_0 = \frac{1}{\sum_{k=0}^{m} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}}$$

$$\pi_m = \frac{\left(\frac{\lambda}{\mu}\right)^m \frac{1}{m!}}{\sum_{k=0}^{m} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}}$$
5 Worst-Case Event Systems [5]

5.1 Competitive Analysis
An online algorithm $A$ is called $c$-competitive if for all finite input sequences $I$ it holds that

$$\text{cost}_A(I) \leq c \cdot \text{cost}_{\text{opt}}(I) + k$$

where $k$ is an input-independent constant. If $k = 0$, $A$ is called strictly $c$-competitive.

5.2 Greedy Algorithm
A greedy will always choose the local optimum in the (most often false) hope to find the global optimum.