Software Engineering

Cicio Ionuț

10/01/2025

Contents

1 Software models 4
1.1 The "Amazon Prime Video" article 4
1.2 Formal theory 5
1.2.1 Markov chain 5
1.2.2 Markov decision process 5
1.2.3 Example 6
1.3 Tips and tricks 7
1.3.1 Average 7
1.3.2 Welford's online algorithm (standard deviation) 7
1.3.3 Euler's method for differential equations 8

2 `C++` 9
2.1 The `random` library 9
2.1.1 `std::default_random_engine` 9
2.1.2 Operator overloading *(quick note)* 10
2.1.3 Distributions 10
2.1.3.1 `uniform_int_distribution` [1] 10
2.1.3.2 `uniform_real_distribution` [2] 12
2.1.3.3 Bernoulli 12
2.1.3.4 Normal 12
2.1.3.5 Exponential 12
2.1.3.6 Poisson 12
2.1.3.7 Geometric 12
2.1.3.8 `discrete_distribution` 12
2.2 Dynamic structures 13
2.2.1 Memory allocation and deallocation 13
2.2.2 `std::vector<T>()` instead of `malloc()` 13
2.2.3 `std::deque<T>()` 13
2.2.4 Sets 13
2.2.5 Maps 13
2.3 I/O 13
2.3.1 `#include <iostream>` 13
2.3.2 Files 13

3 Debugging with `gdb` 14

4 Exercises 15
4.1 First examples 15
4.1.1 A simple Markov chain `[1100]` 15
4.1.2 Markov chains network pt.1 `[1200]` 16
4.1.3 Markov chains network pt.2 `[1300]` 16
4.1.4 Markov chains network pt.3 `[1400]` 17
4.2 Traffic light `[2000]` 17

4.3 Control center 17
4.3.1 No network `[3100]` 17
4.3.2 Network monitor 17
4.3.2.1 No faults `[3200]` 17
4.3.2.2 Faults & no repair `[3300]` 17
4.3.3 Faults & repair `[3400]` 18
4.3.4 Faults & repair + correct protocol `[3500]` 18
4.4 Statistics 18
4.4.1 Expected value `[4100]` 18
4.4.2 Probability `[4200]` 18
4.5 Development process simulation 19
4.5.1 Random transition matrix `[5100]` 19
4.5.2 `[5200]` Software development & error detection 20
4.5.3 Optimizing costs for the development team `[5300]` 23
4.5.4 Key performance index `[5400]` 23
4.6 Complex systems 23
4.6.1 Insulin pump `[6100]` 23
4.6.2 Buffer `[6200]` 23
4.6.3 Server `[6300]` 23

5 Exam **23**
5.1 Development team (time & cost) 23
5.2 Backend load balancing 23
5.2.1 Env 23
5.2.2 Dispatcher, Server and Database 23
5.2.3 Response time 23
5.3 Heater simulation 23

6 MOCC library **24**
6.1 Observer Pattern 24
6.2 `C++` generics & virtual methods 24

7 Extras **25**
7.1 VDM (Vienna Development Method) 25
7.1.1 It's cool, I promise 25
7.1.2 VDM++ to design correct UMLs 25
7.2 Advanced testing techinques (in `Rust`) 25
7.2.1 Mocking (mockall) 25
7.2.2 Fuzzying (cargo-fuzz) 25
7.2.3 Property-based Testing 25
7.2.4 Test Augmentation (Miri, Loom) 25
7.2.5 Performance testing 25
7.2.6 Playwright & UI testing? 25

Bibliography **26**

1 Software models

Software projects require **design choices** that often can't be driven by experience or reasoning alone. That's why a **model** of the project is needed to compare different solutions. In this course, to describe software systems, we use **discrete time Markov chains**.

1.1 The "Amazon Prime Video" article

If you were tasked with designing the software architecture for **Amazon Prime Video** *(a live streaming service for Amazon)*, how would you go about it? What if you had the to keep the costs minimal? Would you use a distributed architecture or a monolith application?

More often than not, monolith applications are considered **more costly** and **less scalable** than the counterpart due to an inefficient usage of resources. But, in a recent article, a Senior SDE at Prime Video describes how they *"**reduced the cost** of the audio/video monitoring infrastructure by **90%**"* [3] by using a monolith architecture.

While there isn't always definitive answer, one way to go about this kind of choice is building a model of the system to compare the solutions. In the case of Prime Video, *"the audio/video monitoring service consists of three major components:"* [3]

- the **media converter** converts input audio/video streams
- the **defect detectors** analyze frames and audio buffers in real-time
- the **orchestrator** controls the flow in the service

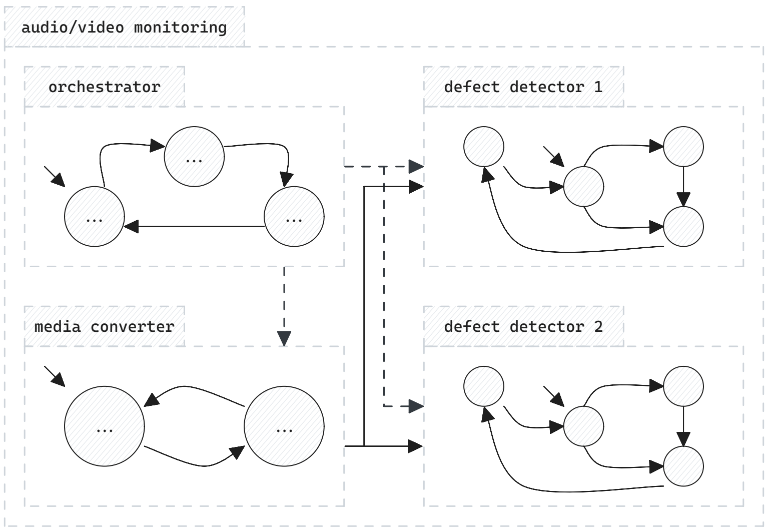


Figure 1: audio/video monitoring system

The system can be **simulated** by modeling its components as **interconnected discrete time Markov chains**.

1.2 Formal theory

1.2.1 Markov chain

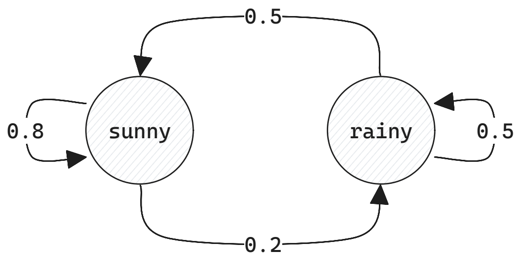
A Markov chain is defined by a set of **states** S and the **transition probability** $p : S \times S \to [0, 1]$ such that $p(s'|s)$ is the probability to transition to state s' if the current state is s, with the constraint that

$$\forall s \in S \ \sum_{s' \in S} p(s'|s) = 1 \tag{1}$$

For instance, the weather can be modeled with $S = \{\text{sunny}, \text{rainy}\}$ and p such that

$p =$

	sunny	rainy
sunny	0.8	0.2
rainy	0.5	0.5



If the chain transitions at discrete time steps, meaning the time steps $t_0, t_1, t_2, \ldots$ are a **countable**, then it's called a DTMC (discrete-time Markov chain), otherwise it's called a CTMC (continuous-time Markov chain).

This kind model is **limited**. To describe complex systems (e.g. servers, rockets, medical devices) the concepts of **input** and **output** are needed.

1.2.2 Markov decision process

A Markov decision process (MDP) is **different** from a Markov chain. A MDP M is a tuple (U, X, Y, p, g) s.t.

- U is the set of **input values**
- X is the set of **states**
- Y is the set of **output values**
- $p : X \times X \times U \to [0, 1]$ is such that $p(x'|x, u)$ is the probability to **transition** from state x to state x' when the **input value** is u
- $g : X \to Y$ is the **output function**
- and let $x_0 \in X$ be the **initial state**

The same constrain in Equation 1 holds for the MDP, with an important difference: **for each input value**, the sum of the transition probabilities for **that input value** must be 1.

$$\forall x \in X \ \ \forall u \in U \ \sum_{x' \in X} p(x'|x, u) = 1 \tag{2}$$

To be more precise, let x_t be the state at the timestep t, then

$$p(x'|x, u) = p(x_{t+1} = x' \mid x_t = x, u_t = u) \tag{3}$$

if the MDP is discrete-time.

1.2.3 Example

The development process of a company can be modeled as a MDP $M = (U, X, Y, p, g)$ s.t.

- $U = \{\varepsilon\}$ because U can't be empty
- $X = \{0, 1, 2, 3, 4\}$
- $Y = \text{Cost} \times \text{Duration}$
- $x_0 = 0$

$$g(x) = \begin{cases} (0, 0) & x = 0 \lor x = 4 \\ (20000, 2) & x = 1 \lor x = 3 \\ (40000, 4) & x = 2 \end{cases} \tag{4}$$

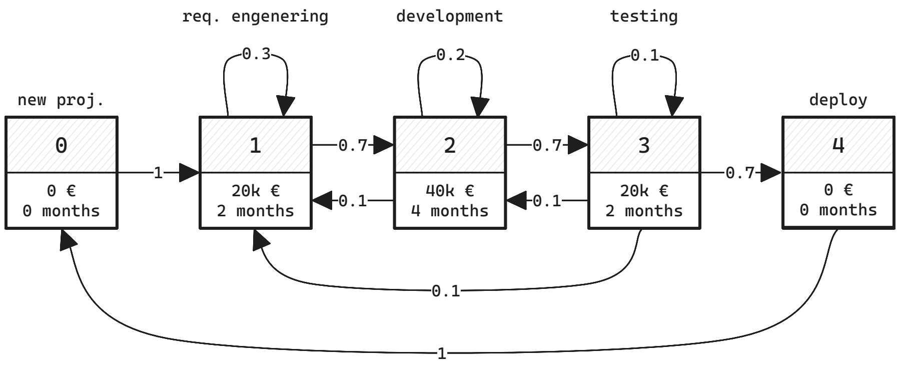


Figure 2: the model of a team's development process

$p =$

ε	**0**	**1**	**2**	**3**	**4**
0	0	1	0	0	0
1	0	.3	.7	0	0
2	0	.1	.2	.7	0
3	0	.1	.1	.1	.7
4	1	0	0	0	0

Notice that we have only 1 table because $|U| = 1$ (we have exactly 1 input value). If we had $U = \{\text{apple}, \text{banana}, \text{orange}\}$ we would have had to describe 3 tables, one **for each input value**.

1.3 Tips and tricks

1.3.1 Average

Given a set of values $X = \{x_1, ..., x_n\} \subset \mathbb{R}$ the average $\overline{x}_n = \frac{\sum_{i=0}^{n} x_i}{n}$ can be computed with a simple procedure

```
float average(std::vector<float> X) {
    float sum = 0;
    for (auto x_i : X)
        sum += x_i;

    return sum / X.size();
}
```

The problem with this procedure is that, by adding up all the values before the division, the **numerator** could **overflow**, even if the value of $\overline{x}_n$ fits within the IEEE-754 limits. There is a way to calculate $\overline{x}_n$ incrementally.

$$\begin{gathered}
\overline{x}_{n+1} = \frac{\sum_{i=0}^{n+1} x_i}{n+1} = \frac{\left(\sum_{i=0}^{n} x_i\right) + x_{n+1}}{n+1} = \frac{\sum_{i=0}^{n} x_i}{n+1} + \frac{x_{n+1}}{n+1} = \\
\frac{\left(\sum_{i=0}^{n} x_i\right) n}{(n+1)n} + \frac{x_{n+1}}{n+1} = \frac{\sum_{i=0}^{n} x_i}{n} \cdot \frac{n}{n+1} + \frac{x_{n+1}}{n+1} = \\
\overline{x}_n \cdot \frac{n}{n+1} + \frac{x_{n+1}}{n+1}
\end{gathered} \tag{5}$$

With this formula the numbers added up are smaller: $\overline{x}_n$ is multiplied by $\frac{n}{n+1} \sim 1$, and if x_{n+1} fits in IEEE-754 then $\frac{x_{n+1}}{n+1}$ can also be encoded.

```
float average_r(std::vector<float> X) {
    float average = 0;
    for (size_t n = 0; n < X.size(); n++)
        average =
            average * ((float)n / (n + 1)) + X[n] / (n + 1);

    return average;
}
```

In `exapmles/average.cpp` the procedure `average_r()` is able to calculate the average and `average()` results in `Inf`.

1.3.2 Welford's online algorithm (standard deviation)

In a similar fashion, it could be faster and require less memory to calculate the **standard deviation** incrementally. Welford's online algorithm can be used for this purpose.

"It is often useful to be able to compute the variance in a single pass, inspecting each value x_i only once; for example, when the data is being collected without enough storage to keep all the values, or when costs of memory access dominate those of computation.” (Wikpedia)

$$\begin{gathered}
M_{2,n} = \sum_{i=1}^{n} (x_i - \overline{x}_n)^2 \\
M_{2,n} = M_{2,n-1} + (x_n - \overline{x}_{n-1})(x_N - \overline{x}_n) \\
\sigma_n^2 = \frac{M_{2,n}}{n} \\
s_n^2 = \frac{M_{2,n}}{n-1}
\end{gathered} \tag{6}$$

1.3.3 Euler’s method for differential equations

Got it from here [4]. Useful if a differential equation can’t be solved analitically.

2 `C++`

This section will cover the basics for the exam.

2.1 The `random` library

The `C++` standard library offers powerful tools to easily implement MDPs.

2.1.1 `std::default_random_engine`

In `C++` there are many ways to **generate random numbers** [5]. Generally it's **not recommended** to use `random()` ① (reasons...). It's recommended to instantiate a **random generator** ④, because it's fast and deterministic (given a **seed**, the sequence of generated numbers is the same). To generate the **seed** a `random_device` is used: it's non deterministic because it uses a **hardware entropy source** (if available) to generate the random numbers. It's also slower.

```
#include <iostream>
#include <random>

int main() {
    std::cout << random() ① << std::endl;

    std::random_device random_device; ②
    std::cout << random_device() ③ << std::endl;

    std::default_random_engine r_engine(random_device() ④ );
    std::cout << r_engine() ⑤ << std::endl;
}
```

Listing 1: `examples/random.cpp`

The typical course of action is to instantiate a `random_device` ②, and use it to generate a seed ④ for a `random_engine`. The other reason random engines are more useful that `random()` is that random engines can be passed to **distributions** (which can be used for MDPs).

From this point on, `std::default_random_engine` will be reffered to as `urng_t` (uniform random number generator type).

```
#include <random>
// works like typedef in C
using urng_t = std::default_random_engine;
int main() {
    std::random_device random_device;
    urng_t urng(random_device());
}
```

2.1.2 Operator overloading *(quick note)*

In Listing 1 in ③ and ⑤, to generate a number, `random_device` and `r_engine` are used like functions with the `()` operator, but they aren't functions, they're instances of a `class`.

That's because in `C++` you can define how a certain operator (like `+`, `+=`, `<<`, `>>`, `[]`, `()` etc..) should behave when used on a instance of the `class`. It's called **operator overloading** (`Java` doesn't have operator overloading, in `Python` it can be done implementing methods with special names [6], like `__add__()` for `+`, in `Rust` it's done by implementing the `Trait` associated to that operation [7]).

For instance `std::cout` is an instance of the `std::basic_ostream` `class`, which overloads the method `operator<<()` [8].

2.1.3 Distributions

Just the capability to generate random numbers isn't enough, we often need to manipulate those numbers to fit our needs. Luckly, `C++` covers **basically all of them**. For example, we can easily simulate the MDP in Figure 2 like this:

```cpp
#include <iostream>
#include <random>

using urng_t = std::default_random_engine;

int main() {
    std::random_device random_device;
    urng_t urng(random_device());

    std::discrete_distribution<> transition_matrix[] = {
        {0, 1},
        {0, .3, .7},
        {0, .2, .2, .6},
        {0, .1, .2, .1, .6},
        {1},
    };

    size_t state = 0;
    for (size_t step = 0; step < 15; step++) {
        state = transition_matrix[state](urng);
        std::cout << state << std::endl;
    }

    return 0;
}
```

Listing 2: `examples/transition_matrix.cpp`

2.1.3.1 `uniform_int_distribution` [1]

Let's consider a simple exercise

> To test a system S it's requried to build a generator that sends value v_t to S every T_t seconds. For each send, the value of T_t is an **integer** chosen uniformly in the range $[20, 30]$.

The `C` code to compute T_t would be `T = 20 + rand() % 11;`, which is very **error prone**, hard to remember and has no semantic value.

In `C++` the same can be done in a **simpler** and **cleaner** way:

```
std::uniform_int_distribution<> random_T(20, 30); ①
size_t T = ② random_T(urng);
```

Now the interval T_t can be easily generated ①, and there's no need to remember any formula or trick. The behaviour of T_t is defined only once ①, so it can be easily changed without introducing bugs or inconsistencies. It's also worth to take a look at the implementation of the exercise above (with the addition that $v_t = T_t$), as it comes up very often in software models.

```
#include <iostream>
#include <random>

using urng_t = std::default_random_engine;

int main() {
    std::random_device random_device;
    urng_t urng(random_device());
    std::uniform_int_distribution<> random_T(20, 30);

    size_t T = random_T(urng), next_request_time = T;
    for (size_t time = 0; time < 1000; time++) {
        if (time < next_request_time)
            continue;

        std::cout << T << std::endl;
        T = random_T(urng);
        next_request_time = time + T;
    }

    return 0;
}
```

Listing 3: `examples/interval_generator.cpp`

The `uniform_int_distribution` has many other uses, for example, it could uniformly generate a random state in a MDP. Let `STATES_SIZE` be the number of states

```
uniform_int_distribution<> random_state(0, STATES_SIZE - 1 ①);
```

`random_state` generates a random state when used. **BE CAREFUL!** Remember to use `STATES_SIZE-1` ①, because `uniform_int_distribution` is **inclusive**... forgettig it can lead to very sneaky bugs: it randomly segfaults at different points of the code. It's very hard to debug unless using `gdb`.

The `uniform_int_distribution` can also generate negative integers, for example $z \in \{z \mid z \in \mathbb{Z} \land z \in [-10, 15]\}$. Its behaviour is **undefined** when the first argument is bigger than the second argument.

2.1.3.2 `uniform_real_distribution` [2]

It's the same as above, with the difference that it generates **real** numbers $\mathbb{R}$, in the range $[a, b)$.

2.1.3.3 Bernoulli

TODO: ... you just have to specify the expected value.

2.1.3.4 Normal

2.1.3.5 Exponential

The Exponential distribution is very useful when simulating user requests (generally, the interval between requests to a servers is described by a Exponential distribution, you just have to specify λ)

2.1.3.6 Poisson

2.1.3.7 Geometric

Does the job

2.1.3.8 `discrete_distribution`

This one is **SUPER USEFUL!**, generates random integers in the range 0, number of items - 1, but it assigns a weight to each item, so each item as a certain weighted probability to be choose. It can be used in transition matrices, and for a bit more complex systems like the status of the project in Listing 9.

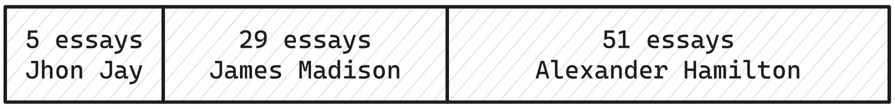


Figure 3: essays authors

2.2 Dynamic structures

2.2.1 Memory allocation and deallocation

If you allocate with `new`, you must deallocate with `delete`, you can't mixup them with `malloc()` and `free()`

2.2.2 `std::vector<T>()` instead of `malloc()`

You don't have to allocate memory, basically never! You just use the structures that are implemented in the standard library, and most of the time they are enough for our use cases. They are really easy to use.

2.2.3 `std::deque<T>()`

2.2.4 Sets

Not needed as much

2.2.5 Maps

Could be useful

2.3 I/O

2.3.1 `#include <iostream>`

2.3.2 Files

3 Debugging with `gdb`

It's super useful! Trust me, if you learn this everything is way easier

4 Exercises

Each exercise has 4 digits `xxxx` that are the same as the ones in the `software` folder in the course material.

4.1 First examples

Now we have to put together our **formal definitions** and our `C++` knowledge to build some simple DTMCs and networks.

4.1.1 A simple Markov chain [1100]

Let's begin our modeling journey by implementing a DTMC M s.t.

- $U = \{\varepsilon\}$ (see Section 1.2.3)
- $X = [0,1] \times [0,1]$, each state is a pair ③ of **real** ① numbers in $[0,1]$
- $Y = [0,1] \times [0,1]$
- $p : X \times X \times U \to X = \mathcal{U}(0,1) \times \mathcal{U}(0,1)$, the transition probability is a **uniform** distribution ②
- $g : X \to Y : (r_0, r_1) \mapsto (r_0, r_1)$ outputs the current state ④
- $X(0) = (0,0)$ ③

```
using real_t ① = double;
const size_t HORIZON = 10;

int main() {
    std::random_device random_device;
    std::default_random_engine random_engine(random_device());
    std::uniform_real_distribution<real_t> uniform(0, 1); ②

    std::vector<real_t> state(2, 0); ③
    std::ofstream log("log");

    for (size_t time = 0; time <= HORIZON; time++) {
        for (auto &r : state) r = uniform(random_engine); ②
        log << time << ' ';
        for (auto r : state) log << r << ' '; t ④
        log << std::endl;
    }

    log.close();
    return 0;
}
```

Listing 4: `software/1100/main.cpp`

4.1.2 Markov chains network pt.1 `[1200]`

In this exercise we build 2 DTMCs M_0, M_1 like the one in the first example Section 4.1.1, with the difference that, and $U_i = [0,1] \times [0,1]$:

- $U_0(t + d) = Y_1(t)$
- $U_1(t + d) = Y_0(t)$

TODO: formula to get input from other stuff and calculate the state..., maybe define

$$\begin{gathered} p : X \times X \times U \to [0,1] \\ (x_0, x_1), (x_0', x_1'), (u_0, u_1) \mapsto \ldots \end{gathered} \tag{7}$$

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

```cpp
const size_t HORIZON = 100;
struct DTMC { real_t state[2]; };

int main() {
    std::vector<DTMC> dtmcs(2, {0, 0});

    for (size_t time = 0; time <= HORIZON; time++) {
        for (size_t r = 0; r < 2; r++) {
            dtmcs[0].state[r] =
                dtmcs[1].state[r] * uniform(random_engine);
            dtmcs[1].state[r] =
                dtmcs[0].state[r] + uniform(random_engine);
        }
    }
}
```

Listing 5: `software/1200/main.cpp`

4.1.3 Markov chains network pt.2 `[1300]`

The same as above, but with a different connection

```cpp
int main() {
    std::vector<DTMC> dtmcs({{1, 1}, {2, 2}});

    for (size_t time = 0; time <= HORIZON; time++) {
        dtmcs[0].state[0] =
            .7 * dtmcs[0].state[0] + .7 * dtmcs[0].state[1];
        dtmcs[0].state[1] =
            -.7 * dtmcs[0].state[0] + .7 * dtmcs[0].state[1];

        dtmcs[1].state[0] =
            dtmcs[1].state[0] + dtmcs[1].state[1];
        dtmcs[1].state[1] =
            -dtmcs[1].state[0] + dtmcs[1].state[1];
        }
    }
```

Listing 6: `software/1300/main.cpp`

4.1.4 Markov chains network pt.3 `[1400]`

The same as above, but with a twist (in the original uses variables to indicate each input... which is sketcy... I can do it with MOCC)

4.2 Traffic light `[2000]`

In this example we want to model a **traffic light**. The three versions of the system on the drive (`2100`, `2200` and `2300`) do the same thing with a different code structure.

```cpp
const size_t HORIZON = 1000;
enum Light { GREEN = 0, YELLOW = 1, RED = 2 };

int main() {
    auto random_timer_duration =
        std::uniform_int_distribution<>(60, 120);

    Light traffic_light = Light::RED;
    size_t timer = random_timer_duration(random_engine);

    for (size_t time = 0; time <= HORIZON; time++) {
        if (timer > 0) {
            timer--;
            continue;
        }

        traffic_light =
            (traffic_light == RED
                ? GREEN
                : (traffic_light == GREEN ? YELLOW : RED));
        timer = random_timer_duration(random_engine);
    }
}
```

Listing 7: `software/2000/main.cpp`

4.3 Control center

4.3.1 No network `[3100]`

4.3.2 Network monitor

4.3.2.1 No faults `[3200]`

4.3.2.2 Faults & no repair `[3300]`

4.3.3 Faults & repair `[3400]`

4.3.4 Faults & repair + correct protocol `[3500]`

4.4 Statistics

4.4.1 Expected value `[4100]`

In this one we just simulate a development process (phase 0, phase 1, and phase 2), and we calculate the average …

4.4.2 Probability `[4200]`

In this one we simulate a more complex software developmen process, and we calculate the average cost (Wait, what? Do we simulate it multiple times?)

4.5 Development process simulation

One of the ways to implement a Markov Chain (like in Section 1.2.1) is by using a **transition matrix**. The simplest implemenation can be done by using a `std::discrete_distribution` by using the trick in Listing 2.

4.5.1 Random transition matrix `[5100]`

In this example we build a **random transition matrix**.

```
const size_t HORIZON = 20, STATES_SIZE = 10;

int main() {
    std::random_device random_device;
    std::default_random_engine random_engine(random_device());
    auto random_state = ①
        std::uniform_int_distribution<>(0, STATES_SIZE - 1);
    std::uniform_real_distribution<> random_real_0_1(0, 1);

    std::vector<std::discrete_distribution<>>
        transition_matrix(STATES_SIZE); ②
    std::ofstream log("log.csv");

    for (size_t state = 0; state < STATES_SIZE; state++) {
        std::vector<real_t> weights(STATES_SIZE); ③
        for (auto &weight : weights)
            weight = random_real_0_1(random_engine);

        transition_matrix[state] = ④
            std::discrete_distribution<>(weights.begin(),
                                         weights.end());
    }

    size_t state = random_state(random_engine);
    for (size_t time = 0; time <= HORIZON; time++) {
        log << time << " " << state << std::endl;
        state = transition_matrix[state ⑤ ](random_engine); ⑥
    }

    log.close();
    return 0;
}
```

Listing 8: `software/5100/main.cpp`

A **transition matrix** is a `vector<discrete_distribution<>>` ② just like in Listing 2. Why can we do this? First of all, the states are numbered from `0` to `STATES_SIZE - 1`, that's why we can generate a random state ① just by generating a number from `0` to `STATES_SIZE - 1`.

The problem with using a simple `uniform_int_distribution` is that we don't want to choose the next state uniformly, we want to do something like in Figure 4.

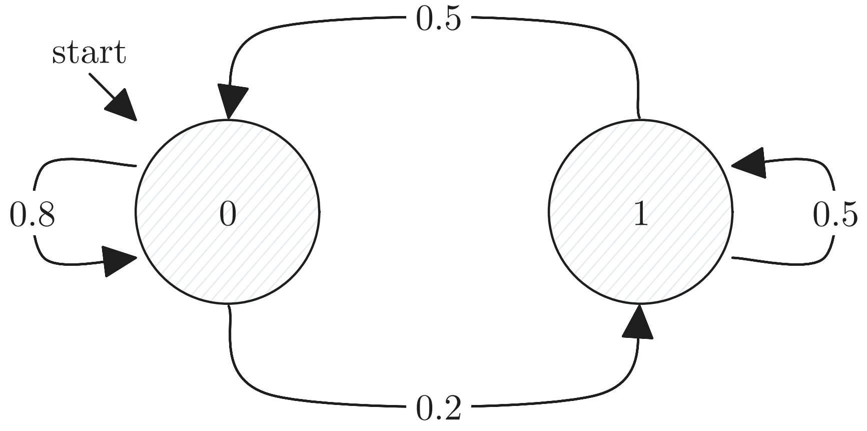


Figure 4: A simple Markov Chain

Luckly for us `std::discrete_distribution<>` does exactly what we want. It takes a list of weights $w_0, w_1, w_2, ..., w_n$ and assigns each index i the probability $p(i) = \frac{\sum_{i=0}^{n} w_i}{w_i}$ (the probability is proportional to the weight, so we have that $\sum_{i=0}^{n} p(i) = 1$ like we would expect in a Markov Chain).

To instantiate the `discrete_distribution` ④, unlike in Listing 2, we need to first calculate the weights ③, as we don't know them in advance.

To randomly generate the next state ⑥ we just have to use the `discrete_distribution` assigned to the current state ⑤.

4.5.2 `[5200]` Software development & error detection

Our next goal is to model the software development process of a team. Each phase takes the team 4 days to complete, and, at the end of each phase the testing team tests the software, and there can be 3 outcomes:

- **no error** is introduced during the phase (we can't actually know it, let's suppose there is an all-knowing "oracle" that can tell us there aren't any errors)
- **no error detected** means that the "oracle" detected an error, but the testing team wasn't able to find it
- **error detected** means that the "oracle" detected an error, and the testing team was able to find it

If we have **no error**, we proceed to the next phase… the same happens if **no error was detected** (because the testing team sucks and didn't find any errors). If we **detect an error** we either reiterate the current phase (with a certain probability, let's suppose 0.8), or we go back to one of the previous phases with equal probability (we do this because, if we find an error, there's a high chance it was introduced in the current phase, and we want to keep the model simple).

In this exercise we take the parameters for each phase (the probability to introduce an error and the probability to not detect an error) from a file.

```
#include <...>

using real_t = double;
const size_t HORIZON = 800, PHASES_SIZE = 3;

enum Outcome ① {
    NO_ERROR = 0,
    NO_ERROR_DETECTED = 1,
    ERROR_DETECTED = 2
};

int main() {
    std::random_device random_device;
    std::default_random_engine urng(random_device());
    std::uniform_real_distribution<> uniform_0_1(0, 1);
    std::vector<std::discrete_distribution<>>
        phases_error_distribution;

    {
        std::ifstream probabilities("probabilities.csv");
        real_t probability_error_introduced,
            probability_error_not_detected;

        while (probabilities >> probability_error_introduced >>
               probability_error_not_detected)
            phases_error_distribution.push_back(
                ② std::discrete_distribution<>({
                    1 - probability_error_introduced,
                    probability_error_introduced *
                        probability_error_not_detected,
                    probability_error_introduced *
                        (1 - probability_error_not_detected),
                }));

        probabilities.close();
        assert(phases_error_distribution.size() ==
               PHASES_SIZE);
    }

    real_t probability_repeat_phase = 0.8;

    size_t phase = 0;
    std::vector<size_t> progress(PHASES_SIZE, 0);
    std::vector<Outcome> outcomes(PHASES_SIZE, NO_ERROR);

    for (size_t time = 0; time < HORIZON; time++) {
        progress[phase]++;

        if (progress[phase] == 4) {
            outcomes[phase] = static_cast<Outcome>(
                phases_error_distribution[phase](urng));
            switch (outcomes[phase]) {
            case NO_ERROR:
            case NO_ERROR_DETECTED:
                phase++;
                break;
            case ERROR_DETECTED:
                if (phase > 0 && uniform_0_1(urng) >
                                     probability_repeat_phase)
                    phase = std::uniform_int_distribution<>(
                        0, phase - 1)(urng);
                break;
            }

            if (phase == PHASES_SIZE)
                break;

            progress[phase] = 0;
        }
    }

    return 0;
}
```

Listing 9: `software/5300/main.cpp`

TODO: `class enum` vs `enum`. We can model the outcomes as an `enum` ①... we can use the `discrete_distribution` trick to choose randomly one of the outcomes ②. The other thing we notice is that we take the probabilities to generate an error and to detect it from a file.

4.5.3 Optimizing costs for the development team `[5300]`

If we want we can manipulate the “parameters” in real life: a better experienced team has a lower probability to introduce an error, but a higher cost. What we can do is:

1. randomly generate the parameters (probability to introduce an error and to not detect it)
2. simulate the development process with the random parameters

By repeating this a bunch of times, we can find out which parameters have the best results, a.k.a generate the lowest development times (there are better techinques like simulated annealing, but this one is simple enough for us).

4.5.4 Key performance index `[5400]`

We can repeat the process in exercise `[5300]`, but this time we can assign a parameter a certain cost, and see which parameters optimize cost and time (or something like that? Idk, I should look up the code again).

4.6 Complex systems

4.6.1 Insulin pump `[6100]`

4.6.2 Buffer `[6200]`

4.6.3 Server `[6300]`

5 Exam

5.1 Development team (time & cost)

5.2 Backend load balancing

5.2.1 Env

5.2.2 Dispatcher, Server and Database

5.2.3 Response time

5.3 Heater simulation

6 MOCC library

Model CheCking

6.1 Observer Pattern

6.2 `C++` generics & virtual methods

TODO...

7 Extras

7.1 VDM (Vienna Development Method)

7.1.1 It's cool, I promise

7.1.2 VDM++ to design correct UMLs

7.2 Advanced testing techinques (in `Rust`)

TODO: cite "Rust for Rustaceans" TODO: unit tests aren't the only type of test

7.2.1 Mocking (mockall)

7.2.2 Fuzzying (cargo-fuzz)

7.2.3 Property-based Testing

7.2.4 Test Augmentation (Miri, Loom)

TODO: Valgrind

7.2.5 Performance testing

TODO: non-functional requirements

7.2.6 Playwright & UI testing?

Bibliography

[1] [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution

[2] [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution

[3] Marcin Kolny, “Scaling up the Prime Video Audio-Video Monitoring Service and Reducing Costs by 90%.” Accessed: Mar. 25, 2024. [Online]. Available: https://web.archive.org/web/20240325042615/https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90#expand

[4] Wikipedia, “Euler method.” [Online]. Available: https://en.wikipedia.org/wiki/Euler_method

[5] “Pseudo-random number generation.” [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random

[6] “operator — Standard operators as functions.” [Online]. Available: https://docs.python.org/3/library/operator.html

[7] “Module ops.” [Online]. Available: https://doc.rust-lang.org/std/ops/index.html#examples

[8] “std::basic_ostream.” [Online]. Available: https://en.cppreference.com/w/cpp/io/basic_ostream/operator_ltlt