EX15

Тимофеенко Никита

31 декабря 2023 г.

Содержание

1. Формулировка задачи

Текст о формулировке задачи.

2. Модуль "config"

Код модуля "config".

3. Модуль "queueing_system"

Код модуля "queueing_system".

4. Модуль "queueing_system_characteristics"

Код модуля "queueing_system_characteristics".

5. Модуль "main"

Код модуля "main".

6. Вывод результатов

Результаты и выводы из работы.

Формулировка задачи

Для приема и обработки донесений от разведгруппы в отделе разведки объединения назначена группа в составе трех офицеров. Ожидаемая интенсивность потока донесений – 30 шт./ч. Каждый офицер может принимать донесения от любой разведгруппы. Освободившийся офицер обрабатывает последнее из поступивших донесений. Интенсивность обработки поступивших донесений одним офицером равна 5 шт./ч. Для оптимизации выполнения поставленной задачи командованием определено ограничение на количество ожидающих донесений в размере 3 шт. Определить характеристики данной СМО.

Модуль "config"

Создание конфигурационного модуля для хранения начальных данных.

```rust
use lazy_static::lazy_static;
use std::sync::Arc;

#[derive(Debug)]
pub struct Config {
    pub num_channels: i32,
    pub queue_size: i32,
    pub lambda_rate: i32,
    pub mu_rate: i32,
    pub initial_state: Arc<Vec<(&'static str, i32)>>,
    pub time: i32,
    pub num_iterations: i32,
    pub step_size: f64
}

lazy_static! {
    pub static ref QUEUING_SYSTEM_CONFIG: Config = Config {
        num_channels: 3,
        queue_size: 3,
        lambda_rate: 30,
        mu_rate: 5,
        initial_state: Arc::new(vec![
            ("S_0", 1),
            ("S_1", 0),
            ("S_2", 0),
            ("S_3", 0),
            ("S_4", 0),
            ("S_5", 0),
            ("S_6", 0),
        ]),
        time: 1,
        num_iterations: 100,
        step_size: 0.01
    };
}
```

Модуль "queueing_system"
Создание модуля с СМО.

```
use std::sync::Arc;
use nalgebra::{DMatrix, DVector};
use std::cmp::Ordering::{Equal, Greater, Less};

use plotters::prelude::*;
use crate::config::{QUEUING_SYSTEM_CONFIG};


pub struct QueuingSystem {
    pub lambda_rate: i32, // Интенсивность потока заявок
    pub mu_rate: i32,     // Интенсивность обработки одним офицером
    pub num_channels: i32, // Количество офицеров
    pub queue_size: i32, // Ограничение на размер очереди
    pub initial_state: Arc<Vec<(&'static str, i32)>>, // Начальное состояние
    pub time: i32, // Время
    pub num_iterations: i32, // Количество итерации
    pub step_size: f64 // Шаг
}

impl QueuingSystem {
    pub fn new(lambda_rate: i32,
               mu_rate: i32,
               num_channels: i32,
               queue_size: i32,
               initial_state: Arc<Vec<(&'static str, i32)>>,
               time: i32,
               num_iterations: i32,
               step_size: f64

    ) -> QueuingSystem {

        QueuingSystem {
            lambda_rate,
            mu_rate,
            num_channels,
            queue_size,
            initial_state,
            time,
            num_iterations,
            step_size
        }
    }
```

```rust
pub fn generate_kolmogorov_matrix(&self) -> Vec<Vec<i32>> {
    let lambda_rate = self.lambda_rate;
    let mu_rate = self.mu_rate;
    let num_channels_i32 = self.num_channels;
    let num_channels_usize = self.num_channels as usize;
    let queue_size = self.queue_size as usize;
    let queue_max_index = num_channels_usize + queue_size;
    let number_of_states = queue_max_index + 1;

    (0..number_of_states).map(|i| {
        (0..number_of_states).map(|j| {
            match i.cmp(&j) {
                Equal => match i {
                    0 => - lambda_rate,
                    _ if i < num_channels_usize => - (lambda_rate + i as i32
                    _ => - (lambda_rate + num_channels_i32 * mu_rate),
                },
                Less => match j {
                    j if j == i + 1 => if i < num_channels_usize { (i as i32
                    _ => 0,
                },
                Greater => if j == i - 1 { lambda_rate } else { 0 },
            }
        }).collect()
    }).collect()
}

fn initial_state_to_dvector(initial_state: Arc<Vec<(&'static str, i32)>>) ->
    let values: Vec<f64> = initial_state
        .iter()
        .map(|(_key, value)| *value as f64)
        .collect();

    DVector::from_vec(values)
}

// Функция для преобразования Vec<Vec<i32>> в DMatrix<f64>
fn kolmogorov_matrix_to_dmatrix(matrix: Vec<Vec<i32>>) -> DMatrix<f64> {
    let rows = matrix.len();
    let cols = matrix.first().map_or(0, Vec::len);

    let flat_matrix: Vec<f64> = matrix.into_iter()
        .flatten()
        .map(|val| val as f64)
        .collect();
```

```
        DMatrix::from_row_slice(rows, cols, &flat_matrix)
    }

    // Функция f(t, x), возвращающая производную состояния
    fn f(_t: f64, state: &DVector<f64>, matrix: &DMatrix<f64>) -> DVector<f64> {
        matrix * state
    }

    // Метод Рунге-Кутты 4-го порядка для одного шага
    fn runge_kutta4_step(&self, state: &DVector<f64>, matrix: &DMatrix<f64>, t:
        let k1 = Self::f(t, state, matrix);
        let k2 = Self::f(t + dt / 2.0, &(state + &k1 * (dt / 2.0)), matrix);
        let k3 = Self::f(t + dt / 2.0, &(state + &k2 * (dt / 2.0)), matrix);
        let k4 = Self::f(t + dt, &(state + &k3 * dt), matrix);

        let new_state = state + &k1 * (dt / 6.0) + &k2 * (dt / 3.0) + &k3 * (dt

        // Нормализация нового состояния
        let sum: f64 = new_state.iter().sum();
        new_state / sum
    }


    // Интегрирование системы уравнений
    pub fn integrate_system(&self) -> Vec<DVector<f64>> {
        let matrix = Self::kolmogorov_matrix_to_dmatrix(self.generate_kolmogorov
        let initial_state_vec = Self::initial_state_to_dvector(Arc::clone(&QUEUI
        let delta_t = QUEUING_SYSTEM_CONFIG.step_size;

        std::iter::successors(Some((initial_state_vec, 0.0)), |(last_state, t)|
            Some((self.runge_kutta4_step(last_state, &matrix, *t, delta_t), t +
        })
            .take((QUEUING_SYSTEM_CONFIG.num_iterations + 1) as usize)
            .map(|(state, _)| state)
            .collect()
    }

}
```

Модуль "queueing_system_characteristics"
Создание модуля с характеристиками СМО.

```rust
use std::collections::BTreeMap;
use crate::queuing_system::QueuingSystem;

pub trait QueuingSystemCharacteristics {
    fn calculate_load_factor(&self) -> f64;
    fn calculate_probability_of_downtime(&self) -> f64;
    fn factorial(n: u64) -> u64;
    fn calculate_probabilities(&self) -> BTreeMap<String, f64>;
    fn calculate_queue_probabilities(&self) -> BTreeMap<String, f64>;
    fn calculate_rejection_probability(&self) -> f64;
    fn calculate_average_incoming_requests_during_t(&self) -> i32;
    fn calculate_average_service_time_per_request(&self) -> f64;
    fn average_service_time_per_channel_for_t(&self) -> f64;
    fn calculate_average_busy_channels(&self) -> f64;
    fn calculate_average_number_of_requests_in_queue(&self) -> f64;
    fn calculate_average_waiting_time_in_queue(&self) -> f64;
    fn calculate_total_number_of_requests(&self) -> f64;
    fn calculate_average_waiting_time(&self) -> f64;
    fn calculate_average_time_in_system(&self) -> f64;
}


impl QueuingSystemCharacteristics for QueuingSystem {
    /// 1
    /// Вычисляет коэффициент загрузки СМО.
    /// # Возвращаемое значение
    /// Коэффициент загрузки системы СМО, тип: `f64`.
    fn calculate_load_factor(&self) -> f64 {
        self.lambda_rate as f64 / self.mu_rate as f64
    }


    /// 2
    /// Вычисляет вероятность простоя системы (P0).
    /// # Возвращаемое значение
    /// Вероятность простоя системы, тип: `f64`.
    fn calculate_probability_of_downtime(&self) -> f64 {
        let ksi = self.calculate_load_factor();
        let sum1: f64 = (0..self.num_channels)
            .map(|i| ksi.powi(i) / Self::factorial(i as u64) as f64)
            .sum();
        let sum2: f64 = (self.num_channels..=(self.num_channels + self.queue_siz
            .map(|i| (self.num_channels.pow(self.num_channels as u32) as f64 / S
```

```
            * (ksi / self.num_channels as f64).powi(i))
        .sum();
    1.0 / (sum1 + sum2)
}

/// Вычисляет факториал числа.
/// # Параметры
/// * `n` - Число, для которого вычисляется факториал.
/// # Возвращаемое значение
/// Факториал заданного числа, тип: `u64`.
fn factorial(n: u64) -> u64 {
    (1..=n).product()
}

/// 3
/// Вероятность того, что i каналов заняты и нет очереди.
/// # Возвращаемое значение
/// Ключ(состояние системы) и значение(вероятность этого состояния), тип: `B
fn calculate_probabilities(&self) -> BTreeMap<String, f64> {
    let p0 = self.calculate_probability_of_downtime();
    let ksi = self.calculate_load_factor();

    (0..=self.num_channels)
        .map(|i| (format!("P_{}", i), p0 * ksi.powi(i) / Self::factorial(i a
        .collect()
}

/// 4
/// Вероятность того, что все s каналов заняты и очередь длины i.
/// # Возвращаемое значение
/// Ключ(состояние системы с очередью) и значение(вероятность этого состояни
fn calculate_queue_probabilities(&self) -> BTreeMap<String, f64> {
    let p0 = self.calculate_probability_of_downtime();
    let ksi = self.calculate_load_factor();
    let s = self.num_channels as f64;
    let n = self.queue_size;

    (1..=n).map(|i| {
        let pi = p0 * (s.powf(s) / Self::factorial(s as u64) as f64) * (ksi
        (format!("P{}", i), pi)
    }).collect()
}

/// 5
/// Вероятность отказа не попасть в очередь длины n, все каналы заняты и оче
/// # Возвращаемое значение
```

```rust
/// Вероятность отказа, когда все каналы заняты и очередь достигла максималь
fn calculate_rejection_probability(&self) -> f64 {
    let p0 = self.calculate_probability_of_downtime();
    let ksi = self.calculate_load_factor();
    let s = self.num_channels as f64;
    let n = self.queue_size;

    p0 * (s.powf(s) / Self::factorial(s as u64) as f64) * (ksi / s).powi(n +
}

/// 6
/// Вычисляет среднее количество заявок, поступающих в систему за время T.
/// # Возвращаемое значение
/// Среднее количество заявок за указанный период времени, тип: `i32`.
fn calculate_average_incoming_requests_during_t(&self) -> i32 {
    self.lambda_rate * self.time
}

/// 7
/// Вычисляет среднее время обслуживания одной заявки.
/// # Возвращаемое значение
/// Среднее время, необходимое для обслуживания одной заявки, тип: `f64`.
fn calculate_average_service_time_per_request(&self) -> f64 {
    1.0 / self.mu_rate as f64
}

/// 8
/// Вычисляет среднее время обслуживания одним каналом заявок, поступивших з
/// # Возвращаемое значение
/// Среднее время обслуживания заявок одним каналом за время T, тип: `f64`.
fn average_service_time_per_channel_for_t(&self) -> f64 {
    let ksi = self.calculate_load_factor();
    ksi * self.time as f64
}

/// 9
/// Вычисляет среднее число занятых каналов в системе.
/// # Возвращаемое значение
/// Среднее количество занятых каналов в системе, тип: `f64`.
fn calculate_average_busy_channels(&self) -> f64 {
    let probabilities = self.calculate_probabilities();
    let queue_probabilities = self.calculate_queue_probabilities();

    probabilities.iter()
        .chain(queue_probabilities.iter())
        .fold(0.0, |acc, (key, prob)| {
```

```
            let channel_count = key.strip_prefix("P")
                .and_then(|num| num.parse::<usize>().ok())
                .unwrap_or(0);
            acc + channel_count as f64 * prob
        })
}

/// 10
/// Вычисляет среднее количество заявок в очереди.
/// # Возвращаемое значение
/// Среднее количество заявок в очереди, тип: `f64`.
fn calculate_average_number_of_requests_in_queue(&self) -> f64 {
    let ksi = self.calculate_load_factor();
    let p0 = self.calculate_probability_of_downtime();
    let s = self.num_channels as f64;

    let numerator = ksi.powf(s + 1.0);
    let denominator = Self::factorial((s - 1.0) as u64) as f64 * (s - ksi).p

    (numerator / denominator) * p0
}

/// 12
/// Вычисляет среднее время пребывания заявки в очереди.
/// # Возвращаемое значение
/// Среднее время пребывания заявки в очереди, тип: `f64`.
fn calculate_average_waiting_time_in_queue(&self) -> f64 {
    let average_number_of_requests_in_queue = self.calculate_average_number_
    let lambda = self.lambda_rate as f64;

    average_number_of_requests_in_queue / lambda
}

/// 13
/// Вычисляет общее количество заявок в системе.
/// # Возвращаемое значение
/// Общее количество заявок в системе, тип: `f64`.
fn calculate_total_number_of_requests(&self) -> f64 {
    let average_number_of_requests_in_queue = self.calculate_average_number_
    let ksi = self.calculate_load_factor();

    average_number_of_requests_in_queue + ksi
}

/// 14
/// Вычисляет среднее время ожидания заявки в системе.
```

```rust
    /// # Возвращаемое значение
    /// Среднее время ожидания заявки в системе, тип: `f64`.
    fn calculate_average_waiting_time(&self) -> f64 {
        let average_number_of_requests_in_queue = self.calculate_average_number_
        let lambda = self.lambda_rate as f64;

        average_number_of_requests_in_queue / lambda
    }

    /// 15
    /// Вычисляет среднее время пребывания заявки в системе.
    /// # Возвращаемое значение
    /// Среднее время пребывания заявки в системе, тип: `f64`.
    fn calculate_average_time_in_system(&self) -> f64 {
        let total_number_of_requests = self.calculate_total_number_of_requests()
        let lambda = self.lambda_rate as f64;

        total_number_of_requests / lambda
    }
}
```

Модуль "main"
Вызов всех методов.

```
use std::sync::Arc;
use crate::config::QUEUING_SYSTEM_CONFIG;
use crate::queuing_system::QueuingSystem;
use crate::queuing_system_characteristics::QueuingSystemCharacteristics;

mod config;
mod queuing_system;
mod queuing_system_characteristics;


fn main() {

    let queuing_system = QueuingSystem::new(
        QUEUING_SYSTEM_CONFIG.lambda_rate,
        QUEUING_SYSTEM_CONFIG.mu_rate,
        QUEUING_SYSTEM_CONFIG.num_channels,
        QUEUING_SYSTEM_CONFIG.queue_size,
        Arc::clone(&QUEUING_SYSTEM_CONFIG.initial_state),
        QUEUING_SYSTEM_CONFIG.time,
        QUEUING_SYSTEM_CONFIG.num_iterations,
        QUEUING_SYSTEM_CONFIG.step_size
    );
    queuing_system.plot_state_graph().expect("Failed to plot state graph");

    let matrix = queuing_system.generate_kolmogorov_matrix();
    println!("Правые части уравнений Колмогорова: {:?}", matrix);

    queuing_system.plot_states(states).expect("Failed to plot states");

    let load_factor = queuing_system.calculate_load_factor();
    let probability_of_downtime = queuing_system.calculate_probability_of_downti
    let probabilities = queuing_system.calculate_probabilities();
    let queue_probabilities = queuing_system.calculate_queue_probabilities();
    let rejection_probability = queuing_system.calculate_rejection_probability()
    let average_incoming_requests_during_t = queuing_system.calculate_average_in
    let average_service_time_per_request = queuing_system.calculate_average_serv
    let average_busy_channels = queuing_system.calculate_average_busy_channels()
    let average_number_of_requests_in_queue = queuing_system.calculate_average_n
    let average_waiting_time_in_queue = queuing_system.calculate_average_waiting
    let total_number_of_requests = queuing_system.calculate_total_number_of_requ
    let average_waiting_time = queuing_system.calculate_average_waiting_time();
    let average_time_in_system = queuing_system.calculate_average_time_in_system
```

```
    println!("Коэффициент загрузки СМО: {}", load_factor);
    println!("Вероятность простоя системы: {}", probability_of_downtime);
    println!("Вероятности того, что i  каналов заняты и нет очереди: {:?}", prob
    println!("Вероятности того, что все s каналов заняты и очередь длины i: {:?}
    println!("Вероятность отказа не попасть в очередь длины n, все каналы заняты
    println!("Среднее число заявок, поступающих за время T: {}", average_incomin
    println!("Среднее время обслуживания заявки: {}", average_service_time_per_r
    println!("Среднее число занятых каналов: {}", average_busy_channels);
    println!("Среднее число заявок в очереди: {}", average_number_of_requests_in
    println!("Среднее время пребывания заявки в очереди: {}", average_waiting_ti
    println!("Общее количество заявок в системе: {}", total_number_of_requests);
    println!("Среднее время ожидания заявки в системе: {}", average_waiting_time
    println!("Среднее время пребывания заявки в системе: {}", average_time_in_sy
}
```

Вывод результатов

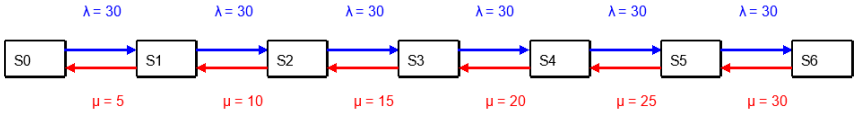

Рис. 1: Граф состояний многоканальной СМО с очередью.

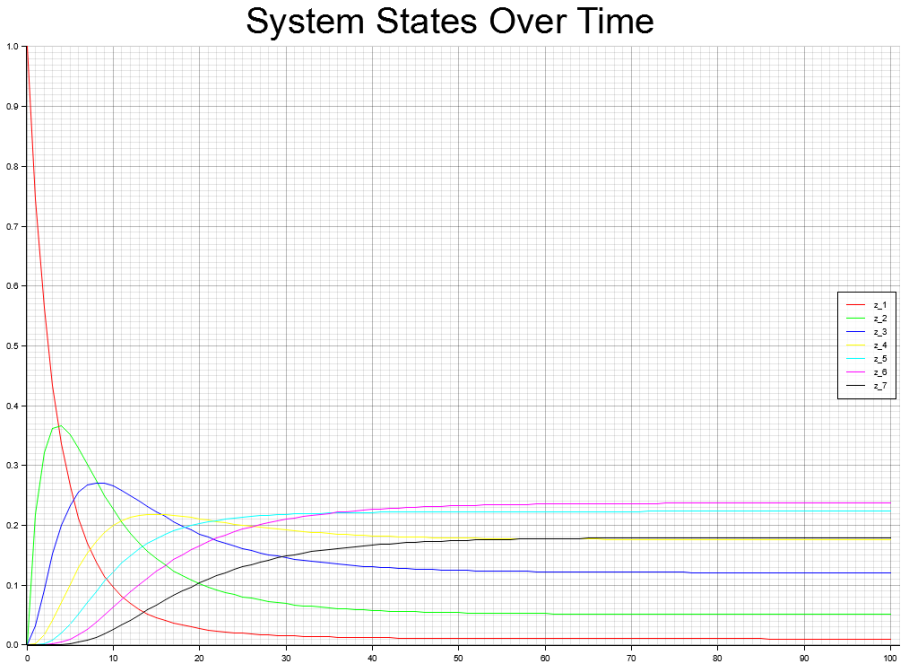

Рис. 2: Динамика многоканальной СМО с очередью.

Описание Системы

В данном документе представлен анализ системы массового обслуживания с использованием уравнений Колмогорова и расчетом основных характеристик системы.

Уравнения Колмогорова

Правые части уравнений Колмогорова для системы представлены в виде матрицы:

$$\begin{pmatrix} -30 & 5 & 0 & 0 & 0 & 0 & 0 \\ 30 & -35 & 10 & 0 & 0 & 0 & 0 \\ 0 & 30 & -40 & 15 & 0 & 0 & 0 \\ 0 & 0 & 30 & -45 & 15 & 0 & 0 \\ 0 & 0 & 0 & 30 & -45 & 15 & 0 \\ 0 & 0 & 0 & 0 & 30 & -45 & 15 \\ 0 & 0 & 0 & 0 & 0 & 30 & -45 \end{pmatrix}$$

Характеристики Системы

Основные характеристики системы представлены ниже:

Коэффициент загрузки СМО: 6
Вероятность простоя системы: 0.0017699115044247787
Вероятности состояний системы:

- Вероятности того, что i каналов заняты и нет очереди:

$$P_0 = 0.0017699115044247787$$
$$P_1 = 0.010619469026548672$$
$$P_2 = 0.03185840707964602$$
$$P_3 = 0.06371681415929203$$

- Вероятности того, что все s каналов заняты и очередь длины i:

$$P1 = 0.12743362831858407$$
$$P2 = 0.25486725663716814$$
$$P3 = 0.5097345132743363$$

Вероятность отказа: не попасть в очередь длины n, все каналы заняты и очередь уже сформирована: 0.5097345132743363
Дополнительные характеристики:

- Среднее число заявок, поступающих за время T: 30
- Среднее время обслуживания заявки: 0.2
- Среднее число занятых каналов: 2.1663716814159293
- Среднее число заявок в очереди: 0.12743362831858407
- Среднее время пребывания заявки в очереди: 0.004247787610619469
- Общее количество заявок в системе: 6.127433628318584
- Среднее время ожидания заявки в системе: 0.004247787610619469
- Среднее время пребывания заявки в системе: 0.20424778761061946