Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

Курсова робота на тему:

ОПТИМАЛЬНІ СТРАТЕГІЇ В ЗАДАЧАХ КЕРУВАННЯ ВИПАДКОВИМИ ПОТОКАМИ В МЕРЕЖІ

Керівник курсової роботи

кандидат фіз.-мат. наук, доцент

Чорней Руслан Костянтинович

_____

(підпис)

Виконав студент 1-го року навчання,

Спеціальності 124 “Системний аналіз”

Случинський Дмитро Юрійович

“____” _____ 2020 р

Київ – 2023

Тема: Оптимальні стратегії в задачах керування випадковими потоками в мережі

Календарний план:

Номер	Назва етапу курсової	Термін виконання етапу	Примітка
1	Отримання теми курсової роботи.	01.11.2022	
2	Опрацювання літератури та джерел	08.03.2023	
3	Розробка моделі	20.03.2023	
4	Програмна реалізація	04.05.2023	
5	Створення графічного інтерфейсу	07.05.2023	
6	Створення презентації	18.05.2023	

ЗМІСТ

ВСТУП

Аналіз випадкових потоків у мережах широко використовується у різних областях, наприклад комп'ютерних мережах[2], електричних мережах[3], керуванні автотрафіку[4], тощо. Одними з типових прикладів мереж, що використовуються, є мережі з циклічними чергами. Тож знаходження оптимального керування для такого типу мереж є важливим. На жаль, складність цієї задачі підвищується зі збільшенням кількості станів, вузлів мереж, вимог. Іншими джерелами труднощів можуть бути функції витрат, які реагують на глобальну поведінку системи. Тому доречніше розбити систему на декілька менших підсистем і досліджувати керування на кожній з них окремо.

Метою даної роботи є розробка програмного забезпечення, що знаходить оптимальну стратегію керування, тобто стратегію, яка призводить до мінімальних середніх витрат на одиницю часу. Об'єктом дослідження є циклічні мережі, що є частинами більших систем. Дослідження спирається на результати описанні статті Р. К. Чорнея, Г. Дадуни та П. С. Кнопова [1]. Предметом дослідження є стратегії керування циклічних мереж.

Робота фокусується на окремій моделі циклічних черг, запропонованій в розділі 4.1 [1]. Постає задача — шляхом аналізу стратегій керування мережею, знайти оптимальну. Пропонується програмна реалізація цієї задачі на мові програмування Python за допомогою ітеративного методу покращення, який описаний у згаданій вище статті.

В першому розділі описуються теоретичні відомості, а також вводяться такі основні поняття: дискретне випадкове поле, Марковське випадкове поле, Марковське випадкове поле з дискретним часом, множина допустимих рішень, допустима локальна стаціонарна Марковська стратегія, імовірнісне ядро переходу, оптимальна стратегія, залежне від часу кероване Марковське випадкове поле та інші. Також в цьому розділі представлена модель циклічної мережі. Вона описана, як скінченний неорієнтований граф, в якому пересувається вказана кількість вимог. Вузлами в графі є нескінченні черги, на початку яких обслуговуються

вимоги. Кожна вимога (одночасно з іншими) з деякою ймовірністю переходять до наступного вузла.

В розділі 2 представляються та описуються реалізовані програмні компоненти для моделей згаданих у попередньому розділі, а також показується метод створення графічного інтерфейсу для застосунку разом з результатами для різних вхідних даних. Вихідний код додається в додатках А і Б відповідно.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

Теоретичний матеріал цього розділу має реферативний характер. Всі використані означення, теореми, наслідки, формули та поняття є запозиченими з статті[1].

1.1 Марковське випадкове поле з дискретним часом

Визначимо структуру взаємодії циклічної мережі скінченним неорієнтованим графом $\Gamma = (V, B)$, де V – множина вершин, B – множина ребер. Позначимо $\{k, j\}$, ребро графа, що сполучає вершини k і j. Околом вершини k є множина вершин $N(k) = \{j: \{k, j\} \in B\}$, тобто множина вершин, з'єднаних ребрами з вершиною k, а її повним околом – $\widetilde{N}(k) = N(k) \cup k$, множина вершин, з'єднаних ребрами з вершиною k включаючи саму вершину k. Для будь-якої підмножини $K \subset V$ визначимо її окіл $N(K) = \bigcup_{k \in K} N(k) - K$ та повний окіл як $\widetilde{N}(K) = N(K) \cup K$.

Нехай $X_i = \{x_i^1, \ldots, x_i^{n_i}\}$ – локальний простір станів у вершині i що належить V. Тоді $X = \times_{i \in V} X_i$ – глобальний простір станів системи. Для кожної підмножини вершин $K \subset V$ вектор $x_K := (x_k, k \in K) \in X_K = \times_{i \in K} X_i$ позначає маргінальний опис стану у вершинах K.

В майбутньому всі введені випадкові величини будуть визначені на сталому ймовірнісному просторі (Ω, F, P).

Означення 1.1.1.

Випадкова величина $\xi: (\Omega, F, P)$, що приймає значення в X називається (дискретним) випадковим полем над $\Gamma = (V, B)$. Випадкові величини, що набувають значень з X_K позначаються ξ_K, а для $K = \{k\}$ ми пишемо ξ_k.

Означення 1.1.2.

Дискретне випадкове поле ξ над $\Gamma = (V, B)$ називається (дискретним) Марковським випадковим полем, якщо виконується наступна умова :

$$P(\xi_k = x_k/\xi_{V-\{k\}} = x_{V-\{k\}}) = P(\xi_k = x_k/\xi_{N(k)} = x_{N(k)}),\ \forall x \in X.$$

Через те, що система змінюється з часом, будемо вважати, що еволюція цієї системи описується тим, що значення x_k з k-ї вершини в кожен момент часу t залежить від попереднього стану всієї системи і тільки через значення вершин її повного околу $\widetilde{N}(k)$ у попередній момент часу $t - 1$. Тобто якщо t приймає дискретні значення $t = 0, 1, \ldots$, то зміна системи описується Марковським процесом з дискретним часом $\xi = (\xi_t : t = 0, 1, \ldots)$.

Означення 1.1.3.

Нехай $\xi = \{\xi_t : t = 0, 1, \ldots\}$ – Марковський процес із дискретним часом і простором станів X для Γ. Тоді процес ξ називається Марковським процесом із синхронними компонентами над (Γ, X), що взаємодіють локально(Марковське поле з дискретним часом), якщо перехідні ймовірності X задовольняють умовам:

локальності : $\forall k \in V,\ x^0, \ldots, x^{t+1} \in X$

$$P(\xi_k^{t+1} = x_k^{t+1}/\xi^t = x^t, \ldots, \xi^0 = x^0) = P(\xi_k^{t+1} = x_k^{t+1}/\xi^t_{\widetilde{N}(k)} = x^t_{\widetilde{N}(k)}) \quad (1)$$

і синхронності : $\forall K \subset V,\ x^0, \ldots, x^{t+1} \in X$

$$P(\xi_K^{t+1} = x_K^{t+1}/\xi^t = x^t) = \prod_{k \in K} P(\xi_k^{t+1} = x_k^{t+1}/\xi^t = x^t) \quad (2)$$

Наслідок 1.1.1(Випливає з (1) та (2))

Для будь-якого моменту часу $t \in N$ можемо знайти механізм переходу з часом для випадкових полів:

$$P(\xi_K^{t+1} = x_K^{t+1} / \xi^t = x^t) = \prod_{k \in K} P(\xi_k^{t+1} = x_k^{t+1} / \xi_{\tilde{N}(k)}^t = x_{\tilde{N}(k)}^t)$$

$$K \subset V, x^t, x^{t+1} \in X$$

1.2 Залежне від часу кероване Марковське випадкове поле

Означення 1.2.1

Загальний набір дій називатимемо $U = \times_{i \in V} U_i$ над Г, де U_i це набір можливих дій (рішень) для особи, яка приймає рішення, у вершині i. U_i є скінченним і не залежить від часу.

Означення 1.2.2

Загальну необмежену історію залежних рішень в час t позначатимемо

$$\Delta^t = \Delta^t\left(x^0, ..., x^t\right) = \left\{\Delta_i^t\left(x^0, ..., x^t\right): i \in V\right\} \in \times_{i \in V} U_i, t = 0, 1, ...$$

Означення 1.2.3

Неприпустимість дій, залежних від часу та стану визначається фіксацією для будь-якої вершини $i \in V$ і будь-якою конфігурацією системи (стану) $x \in X$ набору $U_i^t(x)$ допустимих дій (набір значеннь для допустимого рішення) для Δ_i^t під час t якщо $\xi^t = x$.

Означення 1.2.4

Нехай набір значень контролю U задається як в означенні 1.2.1. Тоді, якщо в будь-який час $t = 0, 1, ...$ рішення Δ_i^t для вузла i прийняте тільки на основі історії повного околу $\tilde{N}(i)$ для i, тобто на основі $x_{\tilde{N}(i)}^0, ..., x_{\tilde{N}(i)}^t$, то послідовність рішень $\delta_i = \left\{\Delta_i^t, t \in N\right\}$ називається локальною стратегією для вершини i.

Означення 1.2.5

Для будь-якої вершини $i \in V$ і стану системи в її повному околі $x_{\underset{\sim}{N(i)}}$ в момент часу t у вузлі i $U_i^t(x_{\underset{\sim}{N(i)}})$ є множиною допустимих рішень для Δ_i^t і в момент t, якщо

$\xi^t_{\sim N(k)} = x_{\sim N(i)}$. Тоді множину допустимих рішень для всієї системи в момент t можемо визначити як $U^t(x) = \times_{i \in V} U^t_i(x_{\sim N(i)})$.

Означення 1.2.6

Допустимою локальною стаціонарною Марковською стратегією δ називатимемо множину функцій $\delta_i = \{\Delta^t_i,\ t \geq 0\}$, якщо для $i \in V$ і будь-яких t, t' виконуються наступні умови:

(локальність)

$$\Delta^t_i = \Delta^t_i(x^0_{\sim N(i)}, \ldots, x^t_{\sim N(i)}) =$$

(Марковість)

$$= \Delta^t_i(x^t_{\sim N(i)}) =$$

(стаціонарність)

$$= \Delta^t_i(x^{t'}_{\sim N(i)}),$$

(допустимість)

$$\Delta^{t'}_i(x_{\sim N(i)}), \in U^t_i(x_{\sim N(i)}),$$

Означення 1.2.7

Пара (ξ, δ) називається керованим Марковським процесом із синхронними компонентами, що взаємодіють локально(залежним від часу керованим Марковським випадковим полем), якщо:

1. $\xi = \{\xi^t : t \geq 0$ є Марковським випадковим полем з дискретним часом із простором станів $X = \times_{i \in V} X_i$;

2. $\delta = (\delta_i : i \in V)$ – допустима локальна стаціонарна Марковська стратегія;

3. Ймовірності переходу ξ визначаються як:

$$P(\xi_S^{t+1} = x_S/\xi^0 = x^0, \Delta^0(\xi^0) = u^0, \ldots, \xi^t = y, \Delta^t(\xi^0, \ldots, \xi^t) = u) =$$

$$= Q_S(x_S/y, u),\ S \subseteq V,\ y \in X,\ u \in U(y),$$

Де $Q_j(x_S/y, u)$ є локально визначеними ядрами переходу, які є інваріантними відносно часу і містять ймовірності переходу між станами системи.

Якщо $S = V$ ми пишемо $Q(x/y, u) = P(\xi^{t+1} = x/\xi^t = y, \Delta^t = u)$.

$$\sum_{x \in X} Q(x/y, u) = 1,\ y \in X$$

1.3 Оптимальна стратегія

Якщо в момент часу $t \in N$ система знаходиться в стані $\xi^t = x^t$ і для прийнятого рішення u^t вона зазнає однокрокових витрат $r(x^t, u^t) \geq 0$. Тоді середні очікувані витрати за час T при початковому $\xi^0 = y$ і стратегії δ будуть дорівнювати:

$$Q_T^{\delta}(y) = E_y^{\delta} \frac{1}{T+1} \sum_{t=0}^{T} r(\xi^t, \Delta^t) := E_y^{\delta} \frac{1}{T+1} \sum_{t=0}^{T} r(\xi^t, \Delta^t(\xi^0, \ldots, \xi^t))).$$

Де E_y^{δ} очікування, пов'язане з керованим процесом (ξ, δ) якщо $\xi^t = y$

Наша задача полягає в тому, щоб знайти стратегію, що мінімізує середні очікувані витрати для T прямує до ∞.

$$R_y^{\delta} = sup Q_T^{\delta}(y) .$$

Позначимо значення, яке ми отримаємо, якщо будемо дотримуватися оптимальної стратегії, як

$$\rho(y) = R_y^{\delta} .$$

Означення 1.3.1.

Стратегію $\delta^* \in LD$(Клас допустимих детермінованих локальних стратегій) називатимемо оптимальною, якщо $\rho(y) = R_y^{\delta^*}$ для всіх $y \in X$.

1.4 Модель циклічної мережі

Як модель для циклічної мережі ми розглядаємо замкнутий цикл окремих вузлів з номерами $1, \ldots, J$, $J \geq 2$, де проводиться деяке обслуговування. Кожен вузол має необмежену кімнату очікування, причому обслуговування відбувається за правилом First-Come-First-Served(Перший прийшов - перший обслуговуєшся). Системою пересуваються K нерозрізненних вимог. $K \geq 2$, і K незмінна. Після того, як вимога покинула вузол j, вона негайно переходить до вузла $j + 1$, причому з умови циклічності випливає що після вузла J іде вузол 1. Можемо описати дану модель як граф $\Gamma(V, B)$, $V = \{1, \ldots, J\}$, $B = \{\{j, j + 1\} : j = 1, \ldots, J\}$, тоді $N(j) = j - 1, j + 1$.

Еволюція системи з часом описується керованим Марковським випадковим полем (ξ, δ) із простором станів $S(K, J) = \{(x_1, \ldots, x_j) \in N^j : x_1 + \cdots + x_j = K\}$. ξ містить довжину черги в кожному вузлі протягом часу роботи системи, тобто $\xi(t) = (\xi_1(t), \ldots, \xi_J(t)) = (x_1, \ldots, x_J)$ показує , що в момент t у вузлі j знаходяться x_J вимог (враховуючи і ту, що вже обслуговується).

Локальні простори рішень $U_i = \{u^1, \ldots, u^i\}$, $i = 0, \ldots, J$ скінченні та незалежні від часу. Тому простір рішень для всієї системи і всіх осіб, що приймають рішення можемо визначити як $U = \times_{i=0,\ldots,J} U_i$.

Посилаючись на теорему 3.1 з [1] можемо знайти єдину оптимальну локальну стратегію δ^* в класі стаціонарних Марковських стратегій, тобто $\delta^* = (\delta_i^*(x_{\tilde{N(i)}}), i = 1, \ldots, J))$. В момент часу t у вузлі j буде прийняте рішення $\Delta_j^t(x_{\tilde{N(j)}})$, що залежить лише від стану околу j. . З цього випливає, що в кінці часового інтервалу у вузлі j обслуговування клієнта закінчується з ймовірністю $p_j(h, u) \in (0, 1)$ і з ймовірністю $q_j(h, u) := 1 - p_j(h, u)$ цей клієнт

залишиться там принаймні на наступний проміжок часу, $h \geq 1, u \in U_i$. Тобто в момент часу t вимога встане в чергу до наступного вузла, або відразу почне обслуговуватися, якщо він не був зайнятий.

Керований процес (ξ, δ) є ергодичним на просторі станів $S(K, J)$ зі стаціонарним розподілом $\pi^{K,J,\delta} := \pi^{\delta} = (\pi^{\delta}(x) : x \in S(K, J))$. Маючи ймовірності переходу вимоги до наступного вузла відносно поточної стратегії δ $p_j(h) = p_j(h, \Delta_j(h))$, можемо знайти π^{δ} для $(x_1, \ldots, x_J) \in S(K, J)$ (Теорема 2.1 [1]):

$$\pi^{K,J}(x_1, \ldots, x_J) = \prod_{j=1}^{J} \left(\frac{\prod_{h=1}^{x_j - 1} q_j(h)}{\prod_{h=1}^{x_j} p_j(h)} \right) G(K, J)^{-1} \quad (3)$$

де $G(K, J)$ – нормувальна константа.

Однокрокові витрати за одиницю часу $t \in N$, під час перебування в стані $\xi^t = x^t$ і прийнятим рішенням u^t дорівнюють $r(x^t, u^t) \geq 0$.

1.5 Ітеративний метод покращення стратегії

Щоб знайти оптимальну стратегію для моделі описаній в розділі 1.5 використаємо ітераційний метод покращення стратегії.

Виберемо деяку стратегію δ та розглянемо невідому функцію $v = (v(x) : x \in S(K, J))$ що задовольняє наступні рівняння:

$$R_y^{\delta} + v(y) = r(y, \delta(y)) + \sum_{x \in S(K,J)} Q(x/y, \delta(y))v(x),\ y \in S(K, J)$$

$$\sum_{x \in S(K,J)} \pi^{\delta}(x)v(x) = 0.$$

Розв'язавши їх для $\{(v(x), R_y^{\delta}) : x, y \in S(K, J)\}$ отримаємо витрати R^{δ}, а також значення $v(x)$ які відповідають кожному зі станів при дотриманні обраної стратегії δ, яка не залежить від початкового стану y.

Тобто для кожного y з простору станів $S(K, J)$, визначимо U_y - набір рішень u в стані y для яких:

$$\sum_{x \in S(K,J)} Q(x/y, u)R_y^{\delta} = R_y^{\delta}$$

$$r(y, u) + \sum_{x \in S(K,J)} Q(x/y, \delta(y))v^{\delta}(x) < R_y^{\delta} + v^{\delta}(y).$$

Якщо даною стратегією δ, ми знайдемо кращу локальну стратегію δ', що складається з $u \in U^y$, відповідних кожному зі станів. Якщо для деякого стану y $U^y = \emptyset$, то відповідне йому рішення u диктує стратегія δ.

Теорема 1.5.1(Теорема 4.1[1])

(1) Якщо $\delta' \neq \delta$, то $R_y^{\delta'} \leq R_y^{\delta}$, $y \in S(K, J)$.

(2) Ітераційний метод покращення стратегії має скінченну кількість ітерацій. Оптимальна стратегія вважається знайденою якщо $U^y = \emptyset$ для всіх y. В цьому випадку $\delta' = \delta^*$ є оптимальною.

2. ПРОГРАМНА РЕАЛІЗАЦІЯ

2.1 Програмні компоненти

Программа була написана на мові програмування Python та з використанням наступних бібліотек:

- Numpy — додає підтримку складних багатошарових масивів і матриць та математичних функцій для операцій з цими масивами.
- Itertools — реалізує складні ітератори.

prob(K,J,U)

Генерує ймовірності pj (h, u).

Аргументи:

- K — кількість вимог у мережі
- J — кількість вузлів у мережі
- U — кількість можливих рішень у вузлі

Повертає:

Тривимірний масив випадкових десяткових чисел [0;1) розміром K * J * U.

states(K, J)

Генерує всі можливі стани системи.

Аргументи:

- K — кількість вимог у мережі
- J — кількість вузлів у мережі

Повертає:

Двовимірний масив, де перший вимір індекс стану, а другий компоненти стану.

decisions(J)

Генерує всі можливі рішення.

Аргументи:

- J — кількість вузлів у мережі

Повертає:

Двовимірний масив, де перший вимір індекс рішення, а другий компоненти рішення.

r(c,b,S,D,J)

Знаходить однокрокові витрати для кожного стану та прийнятого рішення.

Аргументи:

- c,b — масиви для розрахунку однокрокових витрат
- S — масив станів мережі
- D — масив рішень системи
- J — кількість вузлів у мережі

Повертає:

Двовимірний масив, де перший вимір рішення, а другий стани системи.

prod(x,u,p,j)

Функція допомагає в обчислені ергодичного розподілу.

Аргументи:

- x — стан системи
- U — окреме рішення
- p — масив ймовірностей
- j — індекс вузла системи

Повертає:

Верхній і нижній добуток формули (3).

erg_dis(S,D,J,P,strategy)

Обчислює ергодичний розподіл за формулою (3).

Аргументи:

- S — масив станів мережі
- D — масив рішень системи
- J — кількість вузлів у мережі
- P — масив ймовірностей
- strategy — обрана стратегія

Повертає:

Масив десяткових чисел [0;1) довжиною рівною довжині S.

attainable_states(state,D)

Знаходить досяжні стани з окремого стану.

Аргументи:

- state — стан мережі
- D — масив рішень

Повертає:

Три масива. Індекси зайнятих вузлів, можливі рішення та відповідні їм досяжні стани.

possible_decisions(empty_nodes,D)

Функція, що допомагає знайти досяжні стани.

Аргументи:

- empty_nodes — індекси пустих вузлів
- D — масив рішень

Повертає:

Двовимірний масив, де перший індекси рішень, а другий компоненти рішень.

Qs(S,D,P)

Знаходить ядра переходу, для всіх рішень.

Аргументи:

- S — масив станів мережі
- D — масив рішень
- P — масив ймовірностей

Повертає:

Двохвимірний масив розмірністю D × S.

v(Q_s, r_t, pi, strategy)

Знаходить функцію v та загальні витрати для певної стратегії.

Аргументи:

- Q_s — масив з ядрами переходу
- r_t — масив однокрокових витрат
- pi — ергодичний розподіл для заданої стратегії
- strategy — поточна стратегія

Повертає:

Масив, де перше значення загальні витрати, а інші значення v.

OptStrat(S,K,J,D,b,c,P,strategy)

Головна функція. Знаходить оптимальне значення для поточної стратегії.

Аргументи:

- S — масив станів мережі
- K — кількість вимог у мережі
- J — кількість вузлів у мережі
- D — масив рішень
- b,c — масиви для розрахунку однокрокових витрат

- P — масив ймовірностей
- strategy — початкова стратегія

Повертає:

Початкову стратегію, оптимальну стратегію, кількість ітерацій та загальні витрати для оптимальної стратегії.

better_strat(D,S,J,r_t,Q_s,strategy,P)

Функція, що шукає кращі стратегії.

Аргументи:

- D — масив рішень
- S — масив станів мережі
- J — кількість вузлів у мережі
- r_t — масив однокрокових витрат
- Q_s — масив з ядрами переходу
- strategy — поточна стратегія
- P — масив ймовірностей

Повертає:

Кращу стратегію ніж поточна, або поточну, якщо кращої нема.

equal(a,b)

Перевіряє чи масиви однакові.

Аргументи:

- a,b — одновимірні масиви

Повертає:

Значення bull.Якщо масиви рівні True, в іншому випадку False.

2.2 Графічний інтерфейс

Графічний інтерфейс створений на мові програмування Python, а також з використанням віджетів бібліотеки Tkinter.

get_string(inp)

Перетворює строку, яку ввів користувач в список. Обов’язкове розділення через пробіл.

Аргументи:

- inp — рядок, який ввів користувач

Повертає:

Список утворений з рядка.

show_strat_entry()

Показує поле вводу стратегії користувачу.

Аргументів немає. Нічого не повертає.

Generate()

Генерує значення отримавши дані від користувача, або бере тестові. Показує копку “Отримати оптимальну стратегію”. Вводить ймовірності в таблиці. Прив’язана до кнопки “Згенерувати”.

Аргументи немає. Нічого не повертає.

get_strat()

Знаходить оптимальну стратегію. Записує результати функції OptStrat в поля виводу, друкує їх в консолі. Прибирає кнопку “Отримати оптимальну стратегію”. Прив’язана до цієї ж кнопки.

Аргументи немає. Нічого не повертає.

get_list(p)

Перетворює строку в список.

Аргументи:

- p — дана строка

Повертає:

Список отриманий зі строки.

table_input(K,J,U)

Вводить значення в таблицю.

Аргументи:

- K — кількість вимог у мережі
- J — кількість вузлів у мережі
- U — кількість можливих рішень у вузлі

Повертає:

Значення таблиці як масив.

Віджети:

- **Рамки:**
  - main_frame(Основна рамка з усіма іншими віджетами)
  - input_frame(Рамка вводу даних)
  - btn_frame(Рамка з кнопками)
  - tables_frame(Рамка з таблицями)
  - tbl1_frame, tbl2_frame(Власне таблиці)
  - ouput_frame(Рамка виводу)
- **Ярлики**
  - lb_params(“Parameters input”)
  - lb_node_set(“Node set(space split)”)
  - lb_customers(“Number of customers”)

    - lb_alternatives(“Binary alternatives(space split)”)
    - C(“C(space split)”)
    - B(“B(space split)”)
    - stategy(“Custom stategy(space split)”)
    - lb_st_str(“Start strategy”)
    - lb_opt_str(“Optimal strategy”)
    - lb_iters(“Iterasions”)
    - lb_costs(“Cost(R)”)
- **Кнопки**
    - btn_generate(“Generate”)
    - btn_strat(“Get optimal strategy”)
- **Прапорці**
    - test(Для тестування програми. Не видно для користувача.)
    - strategy(Показує поле для стратегії користувача)
- **Поля вводу**
    - node_set_entry(Поле для вводу порядкових номерів вузлів)
    - customers_entry(Поле для вводу кількості вимог)
    - alternatives_entry(Поле для вводу можливих локальних альтернатив)
    - C_entry(Поле для вводу масиву c)
    - B_entry(Поле для вводу масиву b)
    - stategy_entry(Поле для вводу стратегії користувача)
- **Текстові поля**
    - text_st_str(Поле куди програма виводить початкову стратегію)
    - text_opt_str(Поле куди програма виводить оптимальну стратегію)
    - text_iters(Поле куди програма виводить кількість ітерацій)
    - text_costs(Поле куди програма виводить загальні витрати)

2.3 Приклад

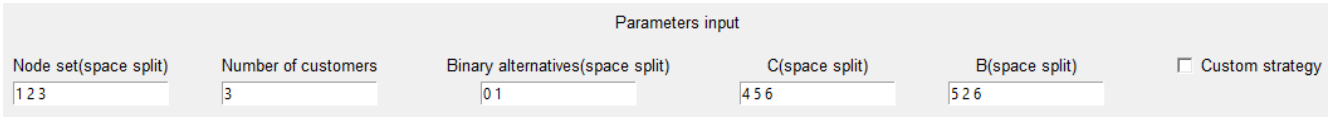
Опишемо структуру моделі множиною вузлів $J = \{1, 2, 3\}$ і $K = 3$ вимогами. Множина допустимих рішень $U_j \equiv \{0, 1\}\ \forall j \in \{1, 2, 3\}$. Рішення у вузлі j позначимо як u_j.

Візьмемо запропоновану функцію однокрокових витрат $r(x, u)$ з прикладу 4.1 [1].

$$r(x, u) = \sum_{j \in S} ((c_j - u_j)x_j + u_j b_j),$$

де $c = (4, 5, 6)$, $b = (5, 2, 6)$.

Введемо дані моделі.



Згенеруємо ймовірності, знайдемо всі можливі стани системи та рішення.

Ймовірності:

p_1(1,0) p_1(1,1)	p_1(2,0) p_1(2,1)	p_1(3,0) p_1(3,1)
p_2(1,0) p_2(1,1)	p_2(2,0) p_2(2,1)	p_2(3,0) p_2(3,1)
p_3(1,0) p_3(1,1)	p_3(2,0) p_3(2,1)	p_3(3,0) p_3(3,1)
0.4097 0.7405	0.3716 0.5728	0.3401 0.2082
0.4927 0.1684	0.5496 0.8462	0.2661 0.6207
0.6737 0.7122	0.7938 0.8182	0.3528 0.5227

Можливі стани системи:

$$x^0 = (0, 0, 3),\ x^1 = (0, 1, 2),\ x^2 = (0, 2, 1),\ x^3 = (0, 3, 0),\ x^4 = (1, 0, 2),$$
$$x^5 = (1, 1, 1),\ x^6 = (1, 2, 0),\ x^7 = (2, 0, 1),\ x^8 = (2, 1, 0),\ x^9 = (3, 0, 0)$$

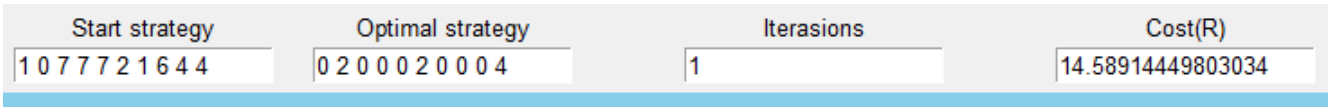
Можливі рішення:

$$u^0 = (0,\ 0,\ 0), u^1 = (0,\ 0,\ 1),\ u^2 = (0,\ 1,\ 0),\ u^3 = (0,\ 1,\ 1),$$
$$u^4 = (1,\ 0,\ 0), u^5 = (1,\ 0,\ 1),\ u^6 = (1,\ 1,\ 0),\ u^7 = (1,\ 1,\ 1)$$

Поле Custom strategy залишимо незаповненим. Після запуску алгоритму отримаємо вектор випадкової початкової стратегії

$\delta = (1,\ 0,\ 7,\ 7,\ 7,\ 2,\ 1,\ 6,\ 4,\ 4)$, де кожен елемент відповідає одному з можливих рішень, а номер елемента — можливому стану системи.

На першій ітерації алгоритм знайшов вектор оптимальної стратегії та відповідне значення середніх очікуваних витрат:



ВИСНОВКИ

Після дослідження циклічних мереж та створення моделі було створено програмне забезпечення, яке, за допомогою ітеративного методу покращення, знаходить оптимальну стратегію керування. Для комфортного використання програми користувачем та отримання результатів було додано графічний інтерфейс.

СПИСОК ЛІТЕРАТУРИ

[1] Ruslan K. Chornei, Hans Daduna V. M., and Pavel S. Knopov. Controlled markov fields with finite state space on graphs. Stochastic Models, 21(4):847–874, 2005.

[2] Ігнатенко О. П. МОДЕЛЮВАННЯ ПРОЦЕСІВ КЕРУВАННЯ КОМП'ЮТЕРНИМИ МЕРЕЖАМИ ЗА УМОВ КОНФЛІКТУ / О. П. Ігнатенко. // Проблеми програмування. 2010. – 2010. – №2. – С. 125–136.

[3] Кулик В. В. Оптимальне керування потоками потужності в неоднорідних електричних мережах з дальніми електропередачами. / В. В. Кулик, С. Я. Вишневський. // Наукові праці Вінницького національного технічного університету. - 2013. - Вип. 2. - Режим доступу: http://nbuv.gov.ua/UJRN/VNTUV_2013_2_2.

[4] Comert, Gurcan & Cetin, Mecit & Begashaw, Negash. (2020). A Simple Traffic Signal Control Using Queue Length Information

ДОДАТКИ

А. Вихідний код func

```
import numpy as np
import random
import itertools
from itertools import product
from itertools import combinations_with_replacement


def prob(K,J,U):
    p = []
    for k in range(K):
        p.append([])
        for j in range(len(J)):
            p[k].append([])
            for u in range(len(U)):
                p[k][j].append(random.random())
    return p


def states(K, J):
    combinations = []
    numbers = range(K+1)
    temp = list(itertools.product(numbers, repeat=J))
    for i in range(len(temp)):
```

```
        if sum(temp[i])==K:
            combinations.append(temp[i])
    return combinations


def decisions(J):
    nodes = [0, 1]
    combinations = itertools.product(nodes, repeat = J)
    return list(combinations)


def r(c,b,S,D,J):
    r = []
    for u in D:
        r_sum = []
        for x in S:
            rsum = 0
            for j in range(len(J)):
                rsum = rsum + (c[j]-u[j])*x[j]+u[j]*b[j]
            r_sum.append(rsum)
        r.append(r_sum)
    return r


def prod(x,u,p,j):
```

```
    p_pr = 1
    q_pr = 1
    for h in range(x[j]):
        if h >=0 :
            p_pr = p_pr*(p[h][j][u[j]])
        else:
            p_pr = p_pr*1
    for h in range(x[j]-1):
        if h >=0 :
            q_pr = q_pr*(1-p[h][j][u[j]])
        else:
            q_pr = q_pr*1
    return p_pr, q_pr


def erg_dis(S,D,J,P,strategy):
    pi = []
    nconst = 0
    for i in range(len(S)):
        x = S[i]
        u = D[strategy[i]]
        up_pr = 1
        dwn_pr = 1
        for j in range(len(J)):
```

```
            up_pr = up_pr*prod(x,u,P,j)[1]
            dwn_pr = dwn_pr*prod(x,u,P,j)[0]
        nconst = nconst + (up_pr/dwn_pr)
        pi.append((up_pr/dwn_pr))
    for i in range(len(pi)):
        pi[i] = pi[i]/nconst
    return pi



def attainable_states(state,D):
    states = []
    full_nodes = []
    empty_nodes = []
    for i in range(len(state)):
        if state[i] != 0:
            full_nodes.append(i)
        else:
            empty_nodes.append(i)
    psbl_dc = possible_decisions(empty_nodes,D)
    for dec in psbl_dc:
        t = []
        for i in state:
            t.append(i)
```

```
        for j in full_nodes:
            if dec[j] == 1:
                t[j] = t[j]-1
                t[(j+1)%len(state)]= t[(j+1)%len(state)]+1
        states.append(t)
    return full_nodes, psbl_dc, states


def possible_decisions(empty_nodes,D):
    decisions = []
    for u in D:
        ind = True
        for i in empty_nodes:
            if u[i] != 0:
                ind = False
        if ind:
            decisions.append(u)
    return decisions


def Qs(S,D,P):
    Qs = []
    for u in D:
        Q = []
        for x in S:
```

```
        Qx = []
        full_nodes, psbl_dc, states = attainable_states(x,D)
        for y in S:
            if list(y) in states:
                p = 1
                idx = states.index(list(y))
                for j in full_nodes:
                    if psbl_dc[idx][j] == 0:
                        p = p*(1-P[x[j]-1][j][u[j]])
                    else:
                        p = p*P[x[j]-1][j][u[j]]
                Qx.append(p)
            else:
                Qx.append(0)
        Q.append(Qx)
    Qs.append(np.array(Q))
return np.array(Qs)


def v(Q_s, r_t, pi, strategy):
    rt =[]
    Qt = []
    for x, u in enumerate(strategy):
        rt.append(r_t[u][x])
```

```
        Qt.append(Q_s[u][x])
    B = np.append(rt, 0)
    A = np.subtract(np.identity(len(strategy)), Qt)
    A = np.insert(A, 0, 1, axis=1)
    A = np.append(A, np.insert(pi, 0, 0))
    A = A.reshape(len(strategy)+1, len(strategy)+1)
    v=np.linalg.solve(A, B)
    return v

def OptStrat(S,K,J,D,b,c,P,strategy):
    strat = strategy.copy()
    start = strategy.copy()
    count = 1
    r_t = r(c,b,S,D,J)
    Q_s = Qs(S,D,P)
    opt_strat = better_strat(D,S,J,r_t,Q_s,strat,P)
    while not equal(opt_strat,strat):
        count = count + 1
        strat = opt_strat
        opt_strat = better_strat(D,S,J,r_t,Q_s,strat,P)
    pi = erg_dis(S,D,J,P,opt_strat)
    v_t = v(Q_s, r_t, pi, opt_strat)
    R = v_t[0]
```

```
    return start, opt_strat, count, R


def better_strat(D,S,J,r_t,Q_s,strategy,P):
    pi = erg_dis(S,D,J,P,strategy)
    v_t = v(Q_s, r_t, pi, strategy)
    bttr_strat = strategy
    actions1 = []
    actions2_r = []
    actions2_l = []
    for y in range(len(S)):
        action1 = []
        actions2_r.append(v_t[0] + v_t[y+1])
        actions2t = []
        for u in range(len(D)):
            action2_l = 0
            sm = 0
            for x in range(len(S)):
                sm = sm + Q_s[u][y][x]*v_t[0]
                action2_l = action2_l + Q_s[u][y][x]*v_t[x+1]
            if sm == v_t[0]:
                action1.append(u)
            actions2t.append(r_t[u][y]+action2_l)
        actions2_l.append(actions2t)
```

```
        actions1.append(action1)
    for y in range(len(S)):
        for u in range(len(D)-1,-1,-1):
            if actions2_l[y][u]<actions2_r[y]:
                for i in range(len(actions1[y])):
                    if actions1[y][i]==u:
                        bttr_strat[y]=u
    return bttr_strat


def equal(a,b):
    try:
        ind = True
        for i in range(len(a)):
            if a[i]!=b[i]:
                ind = False
        return ind
    except:
        return False
```

Б. Вихідний код Main

```
from tkinter import *
from  tkinter import ttk
import tkinter as tk
from func import *


def get_string(inp):
    out = [int(x) for x in inp.split(" ")]
    return out


base = Tk()
base.geometry("1300x500")
base.title('Cyclic Queues')
base['bg'] = '#87CEEB'


main_frame = Frame(base)
main_frame.pack(side=tk.TOP, pady = 5)


lb_params = Label(main_frame, text="Parameters input", width=20,font=("bold",10))
lb_params.pack(side=tk.TOP, pady = 5)


input_frame = Frame(main_frame)
input_frame.pack(side=tk.TOP, pady = 5)
```

```
lb_node_set = Label(input_frame, text="Node set(space split)",
width=20,font=("bold",10))

lb_node_set.grid(row=0,column=0)


lb_customers = Label(input_frame, text="Number of customers",
width=20,font=("bold",10))

lb_customers.grid(row=0,column=1)


lb_alternatives = Label(input_frame, text="Binary alternatives(space split)",
width=30,font=("bold",10))

lb_alternatives.grid(row=0,column=2)


C = Label(input_frame, text="C(space split)", width=20,font=("bold",10))

C.grid(row=0,column=3)


B = Label(input_frame, text="B(space split)", width=20,font=("bold",10))

B.grid(row=0,column=4)


stategy = Label(input_frame, text="Custom stategy(space split)",
width=20,font=("bold",10))

stategy.grid(row=0,column=6)
```

```
node_set_entry = Entry(input_frame,width = 20)
node_set_entry.grid(row=1,column=0)

customers_entry = Entry(input_frame,width = 20)
customers_entry.grid(row=1,column=1)

alternatives_entry = Entry(input_frame,width = 20)
alternatives_entry.grid(row=1,column=2)

C_entry = Entry(input_frame,width = 20)
C_entry.grid(row=1,column=3)

B_entry = Entry(input_frame,width = 20)
B_entry.grid(row=1,column=4)

stategy_entry = Entry(input_frame,width = 20)

def show_strat_entry():
    stategy_entry.grid(row=1,column=6)

test = IntVar()
Checkbutton(input_frame,text="Test option", variable =
test,width=20,font=("bold",10))#.grid(row=0,column=5)
```

```
def Generate():
    global J,K,U,c,b,P,D,S
    if test.get() == 1:
        J = [1,2,3]
        K = 2
        U = [0,1]
        c = [1,2,3]
        b = [3,1,2]
        P = [[[1/3, 1/2],[1/4, 1/2],[1/4, 2/3]],[[2/3, 3/4],[1/3, 2/3],[1/2, 3/4]]]
    else:
        J = get_string(node_set_entry.get())
        K = int(customers_entry.get())
        U = get_string(alternatives_entry.get())
        c = get_string(C_entry.get())
        b = get_string(B_entry.get())
        P = prob(K,J,U)
    S = states(K, len(J))
    D = decisions(len(J))
    btn_strat.pack(side = tk.LEFT,padx = 2, pady = 20)
    table_input(K,J,U)
#    print(J,'\n',K,'\n',U,'\n',c,'\n',b,'\n',P,'\n',S,'\n',D,'\n')
```

```
J = []
K = 0
U = []
c = []
b = []
P = []
S = []
D = []

btn_frame = Frame(main_frame)
btn_frame.pack(side=tk.TOP, pady = 5)

btn_generate=Button(btn_frame, text = "Generate",height = 2, width=20,
bg="darkblue",fg='white',command = lambda : Generate())
btn_generate.pack(side = tk.LEFT,padx = 2, pady = 20)

def get_strat():
    global K,J,D,b,c,P,S
    if strategy.get() == 1:
        strat = get_string(stategy_entry.get())
    else:
        strat = []
        for i in range(len(S)):
```

```
            strat.append(np.random.randint(0, high=len(D)))
    opt = OptStrat(S,K,J,D,b,c,P,strat)
    text_st_str.delete("1.0", END)
    text_opt_str.delete("1.0", END)
    text_iters.delete("1.0", END)
    text_costs.delete("1.0", END)
    print(opt[0],'Початкова стратегія','\n',
      opt[1],'Оптимальна стратегія','\n',
      opt[2],'Кількість кроків','\n',
      opt[3],'R','\n',)
    btn_strat.pack_forget()
    text_st_str.insert("1.0"," ".join(get_list(opt[0])))
    text_opt_str.insert("1.0"," ".join(get_list(opt[1])))
    text_iters.insert("1.0",str(opt[2]))
    text_costs.insert("1.0",str(opt[3]))


def get_list(p):
    lp = []
    for i in range(len(p)):
        lp.append(str(p[i]))
    return lp


def table_input(K,J,U):
```

```
global P
p_text = []
for k in range(len(J)):
    p_text.append([])
    for j in range(K):
        p_text[k].append([])
        for u in range(len(U)):
            p_text[k][j].append('p_'+str(k+1)+'('+str(j+1)+','+str(u)+')')

for item in tb1.get_children():
    tb1.delete(item)
for item in tb2.get_children():
    tb2.delete(item)

tb1['columns'] = [x for x in range(K)]
tb2['columns'] = [x for x in range(K)]
for i in p_text:
    tb1.insert('', tk.END, values=i)
P_tbl = []

p = []
for k in range(K):
    p.append([])
```

```
        for j in range(len(J)):
            p[k].append([])
            for u in range(len(U)):
                p[k][j].append(round(P[k][j][u],4))

    for j in range(len(p[0])):
        P_tbl.append([])

    for i in range(len(p)):
        for j in range(len(p[i])):
            P_tbl[j].append(p[i][j])
    for i in P_tbl:
        tb2.insert('', tk.END, values=i)
    return p_text

btn_strat=Button(btn_frame, text = "Get optimal strategy",height = 2, width=20,
bg="darkblue",fg='white',command = lambda : get_strat())

strategy = IntVar()
Checkbutton(input_frame,text="Custom strategy", variable =
strategy,width=20,font=("bold",10),command = lambda :
show_strat_entry()).grid(row=0,column=6)
tables_frame = Frame(main_frame)
```

```
tables_frame.pack(side=tk.TOP, pady = 5)


tbl1_frame = Frame(tables_frame)

tbl1_frame.pack(side=tk.LEFT,padx = 5, pady = 5,expand=False)

tbl1_frame_scroll = Scrollbar(tbl1_frame)

tbl1_frame_scroll.pack(side= RIGHT,fill=Y)

tb1 = ttk.Treeview(tbl1_frame,yscrollcommand=tbl1_frame_scroll.set, xscrollcommand
=tbl1_frame_scroll.set,show = '')

tb1.pack()

tbl1_frame_scroll.config(command=tb1.xview)


tbl2_frame = Frame(tables_frame)

tbl2_frame.pack(side=tk.RIGHT,padx = 5, pady = 5,expand=False)

tbl2_frame_scroll = Scrollbar(tbl2_frame)

tbl2_frame_scroll.pack(side= RIGHT,fill=Y)

tb2 = ttk.Treeview(tbl2_frame,yscrollcommand=tbl2_frame_scroll.set, xscrollcommand
=tbl2_frame_scroll.set,show = '')

tb2.pack()

tbl2_frame_scroll.config(command=tb2.xview)


ouput_frame = Frame(main_frame)

ouput_frame.pack(side=tk.TOP, pady = 5)
```

```
lb_st_str = Label(ouput_frame, text="Start strategy", width=20,font=("bold",10))
lb_st_str.grid(row=0,column=0)

lb_opt_str = Label(ouput_frame, text="Optimal strategy", width=20,font=("bold",10))
lb_opt_str.grid(row=0,column=1)

lb_iters = Label(ouput_frame, text="Iterasions", width=30,font=("bold",10))
lb_iters.grid(row=0,column=2)

lb_costs = Label(ouput_frame, text="Cost(R)", width=20,font=("bold",10))
lb_costs.grid(row=0,column=3)

text_st_str = Text(ouput_frame, height=1,width=20,font=("bold",10))
text_st_str.grid(row=1,column=0)

text_opt_str = Text(ouput_frame, height=1,width=20,font=("bold",10))
text_opt_str.grid(row=1,column=1)

text_iters = Text(ouput_frame, height=1,width=20,font=("bold",10))
text_iters.grid(row=1,column=2)

text_costs = Text(ouput_frame, height=1,width=20,font=("bold",10))
```

```
text_costs.grid(row=1,column=3)



menubar = Menu(base)

file = Menu(menubar, tearoff = 0)

menubar.add_cascade(label ='Exit', menu = file)

file.add_command(label ='Exit programm', command = base.destroy)

base.config(menu = menubar)


base.mainloop()
```