Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ”КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ”
Кафедра математики факультету інформатики

Курсова робота на тему:

Оптимальні стратегії в задачах керування випадковими потоками в мережі

Керівник курсової роботи:
к. ф.-м. н. *Чорней Р.К.*

_____
(підпис)

”____”_____ 2020 р.

Виконала студентка
3-го року навчання спеціальності
113 ”Прикладна математика”
Гак Софія Володимирівна

Київ – 2020

Тема: _____

Календарний план виконання роботи:

№ п/п	Назва етапу	Термін виконання	Примітка
1.	Отримання завдання на курсову роботу	11.10.2019	
2.	Огляд літератури за темою роботи	18.12.2019	
3.	Розбір задачі	25.02.2020	
4.	Програмна реалізація	09.05.2020	
5.	Створення слайдів для доповіді та написання доповіді	11.05.2020	

Студент _____

Керівник _____

”____”_____

ЗМІСТ

ВСТУП

Нині аналіз випадкових потоків в мережах є надто актуальним. Потоки пакетів даних в комп'ютерній мережі, автомобілей та інших транспортних засобів на дорогах, людей при обслуговуванні в державних установах, медичних закладах і т.п. Особливий інтерес представляє моделювання мереж із чергами (у всіх вище перелічених системах це явище присутнє) та оптимальне керування випадковими потоками в них. Причому часом такі системи можуть бути надто складними, і пошук оптимальної стратегії при врахуванні станів усіх їх складових видається важким завданням. Тому доречніше розбивати такі системи на підсистеми та місцево контролювати потоки в кожній з них.

В цій роботі розглядаються моделі циклічних мереж як частин деякої більшої системи. Матеріал спирається на результати досліджень авторів Р. К. Чорнея, Г. Дадуни та П. С. Кнопова, висвітлених у статті про керовані Марковські поля зі скінченним простором станів на графах (див. [1]). В розділі 1 подається основний теоретичний матеріал, де вводяться поняття керованого марковського поля над графом або керованого Марковського процесу із синхронними компонентами, що взаємодіють локально; ядер переходів - імовірнісних правил, що диктують зміну поведінки системи з часом; описується дві моделі. Циклічна мережа подається у вигляді скінченного неорієнтованого графа, в якому рухається фіксована кількість вимог. Вузли представляють пункти надання деякого сервісу, в кожному з яких може утворюватися черга. Переходи вимог до наступного (попереднього) вузла мають імовірнісний характер та можуть відбуватися синхронно по всій мережі. Довжина черги в кожному вузлі відслідковується багатокроковим процесом, керованим деякою стратегією.

Метою роботи є побудова застосунку, що вирішуватиме основну задачу – пошук такої (оптимальної) стратегії, що мінімізує середні витрати мережі за одиницю часу. Для її вирішення застосовується ітеративний метод покращення стратегії ([1]).

В розділі 2 наводиться опис реалізованих методів у програмі, а також її результати на різних вхідних даних для моделей, описаних у теоретичній частині роботи (вихідний код додається).

1. ТЕОРЕТИЧНА ІНФОРМАЦІЯ

Матеріал цього розділу має реферативний характер. Використовуються теоретичні положення з роботи [1].

1.1. Марковське випадкове поле з дискретним часом

Структуру взаємодії компонент спостережуваної циклічної мережі опишемо скінченним неорієнтованим графом Г = (V, B), де V – множина вершин, B – множина ребер. Нехай $\{k, j\}$ – ребро графа, що з'єднує вершини k і j. *Околом* вершини k назвемо множину вершин $N(k) = \{j : \{k, j\} \in B\}$, а її *повним околом* – $\widetilde{N}(k) = N(k) \cup k$. Для будь-якої підмножини вершин $K \subset V$ визначимо її окіл як $N(K) = \bigcup_{k \in K} N(k) - K$ та повний окіл як $\widetilde{N}(K) = N(K) \cup K$.
Нехай $X_i = \{x_i^1, ..., x_i^{n_i}\}$ – *локальний простір станів* у вершині $i \in V$. Тоді $X = \times_{i \in V} X_i$ – глобальний простір станів системи. Для довільної 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.
Надалі всі введені випадкові величини будуть визначені на фіксованому ймовірнісному просторі $(\Omega, \mathcal{F}, P)$.

Означення 1.1.1. *(Марковське випадкове поле)*
Випадкова величина $\xi : (\Omega, \mathcal{F}, P) \to X$ називається (дискретним) марковським випадковим полем, або марковською мережею, над Г = (V, B), якщо виконується умова Маркова:

$$P(\xi_k = x_k / \xi_{V-\{k\}} = x_{V-\{k\}}) = P(\xi_k = x_k / \xi_{N(k)} = x_{N(k)}), \forall x \in X, \qquad (1)$$

де через ξ_K позначаємо випадкові величини, що набувають значень з X_K, зокрема, для $K = \{k\}$ пишемо ξ_k.

Оскільки ми спостерігаємо за поведінкою системи крізь час, то доречно вважати, що її еволюція – зміна станів – описується послідовністю таких випадкових полів ξ_k^t, що позначають випадкове значення в момент часу t = 0, 1,... для вершини $k \in K$. Інакше кажучи, поведінка системи описується *марковським процесом з дискретним часом* $\xi = (\xi^t : t = 0, 1, ...)$.

Означення 1.1.2. *(Марковське поле з дискретним часом)*
Нехай $\xi = \{\xi^t, t = 0, 1, ...\}$ *– марковський процес із дискретним часом і простором станів* X*. Тоді якщо перехідні ймовірності* X *задовольняють умовам: локальності* : $\forall k \in V, x^0, ..., x^{t+1} \in X$

$$P(\xi_k^{t+1} = x_k^{t+1}/\xi^t = x^t, ..., \xi^0 = x^0) = P(\xi_k^{t+1} = x_k^{t+1}/\xi^t_{\widetilde{N}(k)} = x^t_{\widetilde{N}(k)}) \qquad (2)$$

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

$$P(\xi_K^{t+1} = x_K^{t+1}/\xi^t = x^t) = \prod_{k \in K} P(\xi_k^{t+1} = x_k^{t+1}/\xi^t = x^t), \qquad (3)$$

то процес ξ *і граф* Γ *утворюють Марковське випадкове поле із синхронними компонентами, що взаємодіють локально, коротко – Марковське поле з дискретним часом.*

Наслідок 1.1.2.1. *Об'єднавши умови (2) та (3), можемо описати перехідний механізм спостережуваного процесу для будь-якого моменту часу* $t \in \mathbb{N}$*:*

$$P(\xi_K^{t+1} = x_K^{t+1}/\xi^t = x^t) = \prod_{k \in K} P(\xi_k^{t+1} = x_k^{t+1}/\xi^t_{\widetilde{N}(k)} = x^t_{\widetilde{N}(k)}),$$
$$K \subset V, x^t, x^{t+1} \in X.$$

1.2. Залежне від часу кероване Марковське випадкове поле

Означення 1.2.1. .

(1) Множиною допустимих рішень у вершині i та стані системи в її повному околі $x_{\widetilde{N}(i)}$ в момент часу t над графом Γ називатимемо $U_i^t(x_{\widetilde{N}(i)})$, якщо $\xi^t_{\widetilde{N}(i)} = x_{\widetilde{N}(i)}$. Тоді множину допустимих рішень для всієї системи в момент t визначаємо як $U^t(x) = \times_{i\in V} U_i^t(x_{\widetilde{N}(i)})$.

(2) Допустимою локальною стаціонарною Марковською стратегією δ називатимемо множину функцій $\delta_i = \{\Delta_i^t, t \geq 0\}$,
причому для $i \in V$, будь-яких t, t'
(локальність)

$$\Delta_i^t = \Delta_i^t(x^0_{\widetilde{N}(i)}, \ldots, x^t_{\widetilde{N}(i)}) =$$

(Марковість)

$$= \Delta_i^t(x^t_{\widetilde{N}(i)}) =$$

(стаціонарність)

$$= \Delta_i^{t'}(x_{\widetilde{N}(i)}),$$

(допустимість)

$$\Delta_i^{t'}(x_{\widetilde{N}(i)}) \in U_i^t(x_{\widetilde{N}(i)}),$$

де Δ_i^t – залежне від стану рішення у вершині i в момент часу t; $\{\Delta_i^t, t \geq 0\}$ – історія рішень в i до моменту t.

Означення 1.2.2. *(Залежне від часу кероване Марковське випадкове поле)*
Пара (ξ, δ) називається керованим Марковським процесом із синхронними компонентами, що взаємодіють локально, коротко – залежним від часу керованим Марковським випадковим полем, якщо:

1. *$\xi = (\xi^t : t \geq 0)$ – Марковське випадкове поле 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 $Q_S(x_S/y,u)$ – ядро переходу, що визначає незмінне з часом правило (ймовірність) переходу системи з одного стану в інший. Тобто коли б деяке рішення u не було прийняте при стані системи y, ймовірність переходу до наступного стану буде сталою і залежатиме лише від (y,u). У випадку $S = V : Q(x/y,u) = P(\xi^{t+1} = x/\xi^t = y, \Delta^t = u)$.

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

Зауваження. *Ланцюг перетворень, що пояснюють п.3 з попереднього означення, можна знайти в [1]. Загалом вони випливають з властивостей процесу ξ: Марковості, синхронності, локальності та ін.*

1.3. Постановка задачі

Якщо в момент часу $t \in \mathbb{N}$ система перебуває в стані $\xi^t = x^t$, і приймається рішення u^t, то 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:

$$Q_T^\delta(y) = E_y^\delta \frac{1}{T+1} \sum_{t=0}^{T} r(\xi^t, \Delta^t) \coloneqq E_y^\delta \frac{1}{T+1} \sum_{t=0}^{T} r(\xi^t, \Delta^t(\xi^0, \ldots, \xi^t)). \quad (4)$$

Проблема, яку ми маємо вирішити, пов'язана з пошуком такої стратегії δ, яка мінімізує середні очікувані витрати у випадку нескінченного горизонта ($T \to \infty$):

$$R_y^\delta = \lim_{T \to \infty} Q_T^\delta(y). \quad (5)$$

Позначимо значення, яке маємо отримати, додержуючись оптимальної стратегії, як

$$\rho(y) = \inf_\delta R_y^\delta.$$

Означення 1.3.1. *Стратегію δ^* називатимемо оптимальною, якщо $\rho(y) = R_y^{\delta^*}$ $\forall y \in X$.*

1.4. Модель системи

Спостережувану систему – циклічну мережу – змоделюємо замкненим циклом пронумерованих вузлів $1, \dots, J, J \geq 2$, що ототожнюються із пунктами надання певного сервісу. Кожний такий вузол може містити необмежену чергу, а обслуговування вхідних вимог здійснюється за правилом First-Come-First-Served. Всього системою рухатимуться K (нерозрізненних) вимог, $K \geq 1$. Залишивши вузол j, вимога одразу переходить до вузла $j+1, j = 1, \dots, J$, причому з домовленості про циклічність мережі $J+1 := 1, 1-1 := J$. Описуючи модель в термінах графа $\Gamma(V, B)$, введеного раніше: $V = \{1, \dots, J\}, B = \{\{j, j+1\} : j = 1, \dots, J\}$, тоді $N(j) = j-1, j+1$.

Еволюція системи з часом описується керованим Марковським випадковим полем (ξ, δ) із простором станів $S(K, J) = \{(x_1, \dots, x_j) \in \mathbb{N}^J : x_1 + \dots + x_J = K\}$. ξ відслідковує довжину черги в кожному вузлі протягом часу роботи системи, тобто $\xi(t) = (\xi_1(t), \dots, \xi_J(t)) = (x_1, \dots, x_J)$ вказує на те, що в момент t у вузлі j знаходяться x_j вимог (враховуючи і ту, що вже обслуговується), $j = 1, \dots, J$. Локальні простори рішень $U_i = \{u^1, \dots, u^{n_i}\}, i = 0, \dots, J$ є скінченними та незалежними від часу. Тоді простір рішень по всій системі $U = \times_{i=0,\dots,J} U_i$.

Згідно з **теоремою 3.1** з [1] можемо знайти *єдину* оптимальну локальну стратегію δ^* в класі стаціонарних Марковських стратегій, тобто
$\delta^* = (\delta_i^*(x_{\widetilde{N}(i)}), i = 1, \dots, J))$. Тому в момент часу t у вузлі j буде прийняте рішення $\Delta_j^t(x_{\widetilde{N}(j)})$, що за своєю Марковською природою залежить лише від стану околу j.

Повертаючись до роботи вузлів: якщо в момент часу $t \in \mathbb{N}$ у вузлі j обслуговується вимога і $h - 1 \geq 0$ вимог утворюють чергу до цього вузла, тоді протягом часового відрізка $[t, t+1)$ обслуговування поточної вимоги завершиться з імовірністю $p_j(h, u) \in (0, 1)$ або ж вимога залишиться у вузлі з імовірністю $q_j := 1 - p_j(h, u)$ ще принаймні на один проміжок часу, $h \geq 1, u \in U_j$. Вийшовши з вузла в момент t, вимога або встане в чергу до наступного вузла, або ж почне обслуговуватися одразу – у випадку якщо вузол був незайнятим протягом часу $[t, t+1)$ або в час $t+1$ єдина вимога в ньому перейшла до наступного вузла.

Однокрокові витрати в час $t \in \mathbb{N}$ у стані $\xi^t = x^t$ із прийнятим рішенням є $r(x^t, u^t) \geq 0$. Керований процес (ξ, δ) є ергодичним на просторі станів $S(K, J)$ зі стаціонарним розподілом $\pi^{K,J,\delta} := \pi^\delta = (\pi^\delta(x) : x \in S(K, J))$. Отримавши ймовірності переходу вимоги до наступного вузла відповідно до поточної стратегії δ $p_j(h) = p_j(h, \Delta_j(h))$, можемо знайти π^δ для $(x_1, \dots, x_J) \in S(K, J)$ за формулою (з **теореми 2.1** роботи [1]):

$$\pi^{K,J}(x_1, \dots, x_J) = \prod_{j=1}^{J} \left(\frac{\prod_{h=1}^{x_j - 1} q_j(h)}{\prod_{h=1}^{x_j} p_j(h)} \right) G(K, J)^{-1}, \tag{6}$$

де $G(K, J)$ – нормувальна константа.

1.5. Модифікована модель системи

Розглянемо іншу модель, що відрізняється від попередньої рухом вимог мережею. В ній вузли працюють так само – в момент часу $t \in \mathbb{N}$ вимога або залишилася у вузлі j, або її обслуговування завершилося, і вона вийшла з цього вузла. Причому вихід вимоги може передувати одній із двох подій: вона перейде або до наступного $j+1$-го вузла, або до попереднього – $j-1$-го. Визначимо ймовірності перелічених подій таким чином:

- $p_j^1(h,u)$: завершення обслуговування; перехід до наступного вузла $j+1$;
- $p_j^2(h,u)$: завершення обслуговування; перехід до попереднього вузла $j-1$;
- $p_j^3(h,u)$: вимога залишається в поточному вузлі j;

$$\sum_{i\in\{1,2,3\}} p_j^i(h,u) = 1 \quad \forall j,h,u.$$

Тоді формула (6) ергодичного розподілу процесу ξ набуде вигляду:

$$\pi^{K,J}(x_1,\ldots,x_J) = \prod_{j=1}^{J}\left(\frac{\prod_{h=1}^{x_j-1} p_j^3(h)}{\prod_{h=1}^{x_j}(p_j^1(h)+p_j^2(h))}\right) G(K,J)^{-1} \tag{7}$$

Доведення. $p_j^3(h)$ і $q_j(h)$ моделі з 1.4 є рівними, оскільки позначають ймовірність того, що вимога залишиться у вузлі j, і ця подія може бути реалізована одним шляхом в обох випадках.

В моделі з 1.4 вихід вимоги з вузла j супроводжується єдиною можливою подією – вона переходить у наступний $j+1$-й вузол. В даній же моделі вимога може рухатися як вперед, так і назад. Оскільки ці події несумісні, то $p_j(h) = p_j^1(h) + p_j^2(h)$ як ймовірність того, що, вийшовши з вузла j, вимога перейде або до наступного $j+1$, або до попереднього $j-1$. □

1.6. Алгоритм пошуку оптимальної стратегії

Для знаходження оптимальної стратегії в задачі керування, описаній в попередньому розділі, застосовуватимемо *ітераційний метод покращення стратегії*.

Оберемо деяку стратегію δ та розглянемо для деяких поки що невідомих функцій $v = (v(x) : x \in S(K,J))$ рівняння:

$$\begin{gathered} R_y^\delta + v(y) = r(y,\delta(y)) + \sum_{x\in S(K,J)} Q(x/y,\delta(y))v(x), \quad y \in S(K,J) \\ \sum_{x\in S(K,J)} \pi^\delta(x)v(x) = 0. \end{gathered} \tag{8}$$

Розв'язуючи їх для $\{(v(x), R_y^\delta) : x, y \in S(K, J)\}$, отримуємо відповідні кожному стану значення $v(x)$ та значення витрат R_y^δ при дотриманні стратегії δ, яка, окрім того, не залежить від початкового стану y.
Сам процес покращення стратегії полягає у формуванні для кожного стану $y \in S(K, J)$ відповідної множини U^y рішень u в цьому стані, що задовольняють наступним умовам:

$$\begin{gathered} \sum_{x \in S(K,J)} Q(x/y, u) R_y^\delta = R_y^\delta \\ r(y, u) + \sum_{x \in S(K,J)} Q(x/y, u) v^\delta(x) < R_y^\delta + v^\delta(y). \end{gathered} \tag{9}$$

Якщо поточною стартегією є δ, ми визначаємо нову (кращу) локальну стратегію δ', що складається з $u \in U^y$, відповідних кожному зі станів. Якщо для деякого стану y $U^y = \varnothing$, то відповідне йому рішення u диктує стратегія δ.

Теорема 1.6.1. *[1]*

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

(2) Ітераційний метод покращення стратегії є скінченним алгоритмом. Його робота завершується, коли $U^y = \varnothing \ \ \forall y$. В цьому випадку $\delta' = \delta^*$ є оптимальною.

2. ПРОГРАМНА РЕАЛІЗАЦІЯ

Для реалізації алгоритму було застосовану мову програмування Python та бібліотеку Numpy.

2.1. Опис методів

generateProbabilities(J, K, n, model=1)

Генерує випадкові ймовірності виду $p_j(h, u)$ для симуляції роботи системи (у випадку відсутності заданих ймовірностей).
Аргументи:

- J – кількість вузлів у мережі
- K – кількість вимог у мережі
- n – кількість можливих рішень у вузлі
- *model* - модель, з якою працюємо; за замовчуванням - 1

Повертає:
Масив випадкових десяткових чисел з відрізка [0;1) розміром $J \times K \times n$ (при model=1); Три масиви випадкових десяткових чисел з відрізка [0;1) розміром $J \times K \times n$ (при model=2), відповідні елементи яких утворюють імовірнісний розподіл

get_r(S, U, J, b, c)

Обчислює однокрокові витрати $r(x, u)$ за перебування в одному зі станів множини X та прийняття рішення з множини U. Розраховуються всі можливі комбінації
Аргументи:

- S – простір-масив станів мережі
- U – масив рішень на вузлах мережі
- J – кількість вузлів у мережі
- b, c – допоміжні масиви

Повертає:
Двохвимірний масив цілих чисел. Перший вимір – стани, другий – рішення

getStates(J, K)

Генерує всі можливі стани системи на основі її характеристик (кількості вузлів та вимог)
Аргументи:

- *J* – кількість вузлів у мережі
- *K* – кількість вимог у мережі

Повертає:
Двохвимірний масив станів системи. Перший вимір – індекси станів, другий – компоненти стану

getActions(J)

Генерує всі можливі рішення на вузлах мережі
Аргументи:

- *J* – кількість вузлів у мережі

Повертає:
Двохвимірний масив рішень. Перший вимір – індекси рішень, другий – компоненти рішення

statesUtil(J, K, state, states, idx)

Допоміжна рекурсивна функція для генерації станів системи
Аргументи:

- *J* – кількість вузлів у мережі
- *K* – кількість вимог у мережі
- *state* – стан (масив)
- *states* – масив усіх станів
- *idx* – поточний індекс компоненти стану x

Повертає:
Нічого не повертає; безпосередньо працює із масивами state і states.

actionsAndTransUtil(num, el, array, idx, transitions=False)

Допоміжна рекурсивна функція для генерації рішень на вузлах системи або можливих переходів вимог для стану
Аргументи:

- *num* – розмір масиву el
- *el* – масив розміром *num*
- *array* – масив, що містить *el*
- *idx* – поточний індекс компоненти масиву *el*
- *transitions* - булеве значення, що вказує, генеруємо переходи чи рішення; False за замовчуванням

Повертає:
Нічого не повертає; безпосередньо працює із масивами *el* і *array*

getErgodicDist(S, U, strategy, P1, P2=None, P3=None)

Обчислення ергодичного розподілу за формулою (6) або (7) (у випадку модифікованої моделі)
Аргументи:

- S – простір-масив станів мережі
- U – масив усіх рішень на вузлах мережі
- *strategy* – поточна стратегія (масив індексів рішень з масиву U)
- *P1* - масив ймовірностей виду $p_j(h,u)/p_j^1(h,u)$
- *P2* - масив ймовірностей виду $p_j^2(h,u)$; None за замовчуванням
- *P3* - масив ймовірностей виду $p_j^3(h,u)$; None за замовчуванням

Повертає:
Масив десяткових чисел з відрізка [0;1) розміром, відповідним розміру масиву S

prodOfProbs(j, x, u, P1, P2=None, P3=None)

Допоміжна функція для обчислення стаціонарного розподілу
Аргументи:

- j – поточний індекс компоненти стану x

- x – поточний стан
- u – рішення зі стратегії, що відповідає поточному стану x
- *P1* - масив ймовірностей виду $p_j(h,u)/p_j^1(h,u)$
- *P2* - масив ймовірностей виду $p_j^2(h,u)$; None за замовчуванням
- *P3* - масив ймовірностей виду $p_j^3(h,u)$; None за замовчуванням

Повертає:
Два значення, що є результатами обчислення чисельника та знаменника формули (6) або (7)

getTransitions(busy_num, vacant_nodes, U, model=1)

Генерує можливі переходи вимог. Вибирає з масиву рішень лише ті, що можна реалізувати в поточному стані (при model=1) або окремо генерує переходи, застосовуючи допоміжну функцію actionsAndTransUtil
Аргументи:

- *busy_num* – кількість зайнятих вузлів у стані
- *vacant_nodes* – масив індексів незайнятих вузлів
- *U* – масив усіх рішень на вузлах мережі
- model - модель, з якою працюємо; за замовчуванням - 1

Повертає:
Двохвимірний масив можливих переходів. Перший вимір – масиви переходів, другий – значення -1 (при model=2), 0 або 1, що позначають, які дії робити у відповідному вузлі, щоб перейти в інший стан (0 - залишитися на місці, -1 - перейти в попередній вузол, 1 - перейти в наступний вузол)

getAchievableStates(state, U, model=1)

Формування масиву досяжних станів для поточного стану і моделі
Аргументи:

- *state* – поточний стан
- *U* – масив усіх рішень на вузлах мережі
- model - модель, з якою працюємо; за замовчуванням - 1

Повертає:
Двохвимірний масив досяжних станів для поточного стану. Перший вимір – масиви станів, другий – компоненти стану

statesDict(S)

Формує словник з масиву станів системи
Аргументи:

- S – простір-масив станів мережі

Повертає:
Словник станів (ключ – індекс стану, значення – стан-масив)

getQs(S, U, P1, P2=None, P3=None)

Обчислює ядра переходу для всіх можливих рішень
Аргументи:

- S – простір-масив станів мережі
- U – масив усіх рішень на вузлах мережі
- *P1* - масив ймовірностей виду $p_j(h,u)/p_j^1(h,u)$
- *P2* - масив ймовірностей виду $p_j^2(h,u)$; None за замовчуванням
- *P3* - масив ймовірностей виду $p_j^3(h,u)$; None за замовчуванням

Повертає:
Двохвимірний масив розмірністю $|U| \times |S|$

get_v(Q, r, pi, strategy)

Пошук розв'язку системи рівнянь (8) – значень v та середніх очікуваних витрат R
Аргументи:

- Q – масив із ядрами переходу для всіх можливих рішень
- r – масив вартостей виходу зі стану відповідно до прийнятого рішення для всіх станів та рішень
- pi – ергодичний розподіл станів за поточної стратегії
- *strategy* – поточна стратегія

Повертає:
Масив, що за перший елемент має середні очікувані витрати R; решта елементів – значення v

optimalStrategy(S, U, J, b, c, P1, P2=None, P3=None, strategy=None)

Власне, реалізація алгоритму. Функція обирає випадковим чином початкову стратегію (також є можливість задати свою в аргументах) та ітеративно шукає оптимальну, використовуючи допоміжну функцію
Аргументи:

- S – простір-масив станів мережі
- U – масив усіх рішень на вузлах мережі
- J – кількість вузлів у мережі
- b, c – допоміжні масиви для обчислення витрат $r(x, u)$
- *P1* - масив ймовірностей виду $p_j(h, u)/p_j^1(h, u)$
- *P2* - масив ймовірностей виду $p_j^2(h, u)$; None за замовчуванням
- *P3* - масив ймовірностей виду $p_j^3(h, u)$; None за замовчуванням
- *strategy* – поточна стратегія; None за замовчуванням

Повертає:
Масив індексів рішень з масиву U – початкову стратегію (з метою демонстрації), такої ж структури оптимальну стратегію, кількість ітерацій та дійснозначне число R – мінімальні середні очікувані витрати.

betterStrategy(S, U, r, Q, strategy, P1, P2=None, P3=None)

Допоміжна функція для optimalStrategy; однокрокове покращення поточної стратегії; перевірка умов (9)
Аргументи:

- S – простір-масив станів мережі
- U – масив усіх рішень на вузлах мережі
- r – масив вартостей виходу зі стану відповідно до прийнятого рішення для всіх станів та рішень
- Q – масив із ядрами переходу для всіх можливих рішень
- *strategy* – поточна стратегія
- *P1* - масив ймовірностей виду $p_j(h, u)/p_j^1(h, u)$
- *P2* - масив ймовірностей виду $p_j^2(h, u)$; None за замовчуванням

- *P3* - масив ймовірностей виду $p_j^3(h,u)$; None за замовчуванням

Повертає:
Масив індексів рішень з масиву U – кращу за поточну стратегію (у випадку оптимальності повертає ту саму стратегію)

2.2. Робота алгоритму на прикладах

2.2.1. Для моделі з розділу 1.4

Використаємо дані з **прикладу 4.1** з [1].
Структура моделі описується множиною вузлів $\{1, 2, 3\}$ і $K = 2$ вимогами. Множина допустимих рішень $U_j \equiv \{0,1\}\ \ \forall j \in \{1,2,3\}$. Через u_j позначатимемо рішення у вузлі j. На вході маємо ймовірності $p_j(h,u)$:

$$P = \begin{pmatrix} (p_1(1,0); p_1(1,1)) & (p_1(2,0); p_1(2,1)) \\ (p_2(1,0); p_2(1,1)) & (p_2(2,0); p_2(2,1)) \\ (p_3(1,0); p_3(1,1)) & (p_3(2,0); p_3(2,1)) \end{pmatrix} = \begin{pmatrix} (1/3; 1/2) & (2/3; 3/4) \\ (1/4; 1/2) & (1/3; 2/3) \\ (1/4; 2/3) & (1/2; 3/4) \end{pmatrix}$$

і функцію однокрокових втрат $r(x,u)$ (її було запрограмовано саме в такому вигляді):

$$r(x,u) = \sum_{j \in S}((c_j - u_j)x_j + u_j b_j), \tag{10}$$

де $c = (1,2,3), b = (3,1,2)$.
Використавши метод getStates(J=3, K=2) отримуємо можливі стани мережі $S(2,3)$:

$$x^0 = (0,0,2), x^1 = (0,1,1), x^2 = (0,2,0),$$
$$x^3 = (1,0,1), x^4 = (1,1,0), x^5 = (2,0,0)$$

Метод getActions(J=3, K=2) повертає множину U всіх можливі рішення на вузлах:

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

Маючи всі необхідні початкові дані, можемо застосувати метод optimalStrategy(S=$S(2,3)$, U=U, J=3, b=b, c=c, P=P). Залишаємо значення аргумента strategy за замовчуванням – функція самостійно випадковим чином обере початкову стратегію. В результаті отримуємо такий вектор оптимальної стратегії та відповідне їй значення середніх очікуваних витрат:

$$\delta^* = (1,1,2,1,0,0), \quad R^{\delta^*} \approx 3.848$$

(за 3 ітерації при початковій стратегії δ^0 = (5, 7, 4, 6, 3, 3)).

2.2.2. Для модифікованої моделі з розділу 1.5

Задамо складнішу структуру моделі, додавши в обіг додаткову вимогу. Нехай $J = 3, K = 3$. Множину допустимих рішень, як і в попередній моделі, фіксуємо $U_j \equiv \{0, 1\} \ \ \forall j \in \{1, 2, 3\}$.
Для симуляції роботи мережі згенеруємо ймовірності $P_i = (p_j^i(h, u))_{\{j=1,2,3;h=1,2,3;u=0,1\}}$, $i \in \{1, 2, 3\}$, використавши метод generateProbabilities(J, K, 2, model=2):

$$P_1 = \begin{pmatrix} (0.2414; 0.0925) & (0.3023; 0.2198) & (0.4417; 0.3636) \\ (0.32; 0.2989) & (0.4093; 0.1342) & (0.0984; 0.403) \\ (0.4135; 0.094) & (0.2312; 0.41) & (0.2011; 0.2517) \end{pmatrix}$$

$$P_2 = \begin{pmatrix} (0.2931; 0.474) & (0.5174; 0.3571) & (0.4724; 0.4318) \\ (0.12; 0.2771) & (0.2405; 0.6443) & (0.8688; 0.2245) \\ (0.4279; 0.3419) & (0.15; 0.0863) & (0.4022; 0.2653) \end{pmatrix}$$

$$P_3 = \begin{pmatrix} (0.4655; 0.4335) & (0.1802; 0.4231) & (0.0859; 0.2045) \\ (0.56; 0.4239) & (0.3502; 0.2214) & (0.0327; 0.3724) \\ (0.1586; 0.5641) & (0.6187; 0.5035) & (0.3965; 0.4829) \end{pmatrix}$$

Функцію однокрокових втрат $r(x, u)$ лишаємо такою самою (10), змінивши лише c, b:

$$c = (1, 2, 3, 4, 5), b = (3, 1, 2, 2, 1)$$

За допомогою getStates(J=3, K=3) генеруємо стани $S(3, 3)$:

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

З getActions(J=3) отримуємо такі можливі рішення на вузлах:

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

Застосовуємо optimalStrategy(S=$S(3, 3)$, U=U, J=3, b=b, c=c, P1=P_1, P2=P_2, P3=P_3) і отримуємо:

$$\delta^* = (1, 0, 0, 2, 1, 0, 0, 0, 0, 0), \quad R^{\delta^*} \approx 5.41$$

(за 4 ітерації при початковій стратегії δ^0 = (5, 0, 6, 0, 5, 3, 1, 6, 6, 6)).

ВИСНОВКИ

Під час дослідження ознайомилася з ітеративним алгоритмом пошуку оптимальної стратегії для керування випадковими потоками в мережі. Свою увагу зосередила саме на циклічних мережах, розглянувши два типи моделей, описаних в 1.4 і 1.5. Програмно реалізувала знаходження оптимальної стратегії цим методом для обох моделей.

В подальшому можна змоделювати складніші системи, що містять в собі циклічні складові, і розширити функціонал програми для роботи з ними.

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.

ДОДАТКИ

A. Вихідний код

Блок 1: Для генерації даних

```
import numpy as np
def generateProbabilities(J, K, n, model=1):
  if model==1:
    P = np.array([[np.random.random(n) for k in range(K)] for j in range(J)])
    return P
  else:
    temp = np.array([[[np.random.randint(low=1,high=100, size=3) for u in range(n)]
    for k in range(K)] for j in range(J)])
    sums = temp.sum(axis=3)
    temp1 = temp / sums[:,:,:, np.newaxis]
    P1 = [[[temp1[j][k][u][0] for u in range(n)] for k in range(K)] for j in range(J)]
    P2 = [[[temp1[j][k][u][1] for u in range(n)] for k in range(K)] for j in range(J)]
    P3 = [[[temp1[j][k][u][2] for u in range(n)] for k in range(K)] for j in range(J)]
    return P1, P2, P3

def get_r(S, U, J, b, c):
  r = [[sum([(c[j]-u[j])*x[j]+u[j]*b[j] for j in range(J)]) for u in U] for x in S]
  return np.array(r)

def getStates(J, K):
    state = [False for j in range(J)]
    states = []
    for i in range(K+1):
        state[0] = i
        statesUtil(J, K - i, state, states, 1)
    states = np.reshape(states, (-1, J))
    return states

def statesDict(S):
  d = {}
  for i, e in enumerate(S):
    d[i] = e
  return d

def getActions(J):
  u = [False for i in range(J)]
  U = []
  actionsAndTransUtil(J, u, U, 0)
  U = np.reshape(U, (-1, J))
  return U
```

Блок 2: Допоміжні функції

```
def statesUtil(J, K, state, states, idx):
    if (idx > J or K < 0):
        return
    if (idx == J):
        if(K == 0):
            for i in state:
              states.append(i)
        return

    for i in range(K+1):
        state[idx] = i
        statesUtil(J, K - i, state, states, idx + 1)

def actionsAndTransUtil(num, el, array, idx, transitions=False):
  if idx == num:
      for j in el:
        array.append(j)
      return
  if transitions:
    el[idx] = -1
    actionsAndTransUtil(num, el, array, idx + 1, transitions=True)

  el[idx] = 0
  if transitions:
    actionsAndTransUtil(num, el, array, idx + 1, transitions=True)
  else:
    actionsAndTransUtil(num, el, array, idx + 1)

  el[idx] = 1
  if transitions:
    actionsAndTransUtil(num, el, array, idx + 1, transitions=True)
  else:
    actionsAndTransUtil(num, el, array, idx + 1)
```

Блок 3: Обчислення ергодичного розподілу

```
def getErgodicDist(S, U, strategy, P1, P2=None, P3=None):
  num_states = len(S)
  temp = []
  pi=[]
  norm_const=0
  for i in range(num_states):
    u = U[strategy[i]]
    x = S[i]
    if P2 and P3:
      temp.append(np.prod([prodOfProbs(j, x, u, P1, P2, P3)[0]
                           /prodOfProbs(j, x, u, P1, P2, P3)[1]
                           for j in range(J)]))
```

```
    else:
      temp.append(np.prod([prodOfProbs(j, x, u, P1)[0]
                           /prodOfProbs(j, x, u, P1)[1]
                           for j in range(J)]))
    norm_const+=temp[i]
  for t in temp:
    pi.append(t/norm_const)
  return np.array(pi)

def prodOfProbs(j, x, u, P1, P2=None, P3=None):
  if P2 and P3:
    q_prod = np.prod([P3[j][h1][u[j]] if h1>-1 else 1
                      for h1 in range(x[j]-1)])
    p_prod = np.prod([P1[j][h2][u[j]]+P2[j][h2][u[j]]
                      if h2>-1 else 1 for h2 in range(x[j])])
  else:
    q_prod = np.prod([1-P1[j][h1][u[j]] if h1>-1 else 1
                      for h1 in range(x[j]-1)])
    p_prod = np.prod([P1[j][h2][u[j]] if h2>-1 else 1
                      for h2 in range(x[j])])
  return q_prod, p_prod
```

Блок 4: Можливі переходи вимог та досяжні стани

```
def getTransitions(busy_num, vacant_nodes, U, model=1):
  transitions = []
  if model==1:
    for u in U:
      properTransition = True
      for vac in vacant_nodes:
        if u[vac]!=0:
          properTransition = False
      if properTransition:
        transitions.append(u)
  else:
    t = [False for i in range(busy_num)]
    actionsAndTransUtil(busy_num, t, transitions, 0, transitions=True)
    transitions = np.reshape(transitions, (-1, busy_num))
    for v in vacant_nodes:
      transitions = np.insert(transitions, v, 0, axis=1)
  return np.array(transitions)

def getAchievableStates(state, U, model=1):
  achievable_states = []
  n = len(state)
  busy_nodes = [i for i, e in enumerate(state) if e!=0]
  busy_num = len(busy_nodes)
  vacant_nodes = [i for i, e in enumerate(state) if e==0]
  if model==1:
    transitions = getTransitions(busy_num, vacant_nodes, U)
  else:
```

```
    transitions = getTransitions(busy_num, vacant_nodes, U, model=2)
  for transition in transitions:
    temp = [e for e in state]
    for j in busy_nodes:
      if transition[j] == 1:
        temp[j]-=1
        temp[(j+1)%n]+=1
      if transition[j] == -1:
        temp[j]-=1
        temp[j-1]+=1
    for t in temp:
      achievable_states.append(t)
  achievable_states = np.reshape(achievable_states, (-1, n))
  return busy_nodes, transitions, achievable_states
```

Блок 5: Ядра переходу $Q(x/y,u)$

```
def getQs(S, U, P1, P2=None, P3=None):
  allStates = statesDict(S)
  Qs = []
  for u in U:
    Q=[]
    for idx, x in allStates.items():
      if P2 and P3:
        busy_nodes, transitions, achievableStates =
        getAchievableStates(x, U, model=2)
        transitionProbs = [np.prod([P1[j][x[j]-1][u[j]]
                                    if t[j]==0 else P2[j][x[j]-1][u[j]]
                                    if t[j]==1 else P3[j][x[j]-1][u[j]]
                                    for j in busy_nodes]) for t in transitions]
      else:
        busy_nodes, transitions, achievableStates = getAchievableStates(x, U)
        transitionProbs = [np.prod([1-P[j][x[j]-1][u[j]]
                                    if t[j]==0 else P[j][x[j]-1][u[j]]
                                    for j in busy_nodes]) for t in transitions]
      indices = [idx for s in achievableStates
                 for idx, state in allStates.items()
                 if np.array_equal(state, s)]
      probs = [0 for x in allStates]
      for i in range(len(indices)):
        state_idx = indices[i]
        if probs[state_idx]:
          probs[state_idx]+= transitionProbs[i]
        else:
          probs[state_idx] = transitionProbs[i]
      Q.append(probs)
    Qs.append(Q)
  return np.array(Qs)
```

Блок 6: Обчислення значень v

```
def get_v(Q, r, pi, strategy):
  n = len(strategy)+1
  temp_r =[]
  temp_Q = []
  for x, u in enumerate(strategy):
    temp_r.append(r[x][u])
    temp_Q.append(Q[u][x])
  temp_r = np.append(temp_r, 0)
  A = np.subtract(np.identity(n-1), temp_Q)
  A = np.insert(A, 0, 1, axis=1)
  A = np.append(A, np.insert(pi, 0, 0))
  A = A.reshape(n, n)
  v=np.linalg.solve(A, temp_r)
  return v
```

Блок 7: Пошук оптимальної стратегії

```
def optimalStrategy(S, U, J, b, c, P1, P2=None, P3=None, strategy=None):
  if strategy:
    strat = strategy
  else:
    strat = np.random.randint(0, high=len(U), size=len(S))
  init_strategy = strat
  iter_count=1
  all_rs = get_r(S, U, J, b, c)
  if P2 and P3:
     all_Qs = getQs(S, U, P1, P2, P3)
     optimum = betterStrategy(S, U, all_rs, all_Qs, strat, P1, P2, P3)
  else:
    all_Qs = getQs(S, U, P1)
    optimum = betterStrategy(S, U, all_rs, all_Qs, strat, P1)
  while not np.array_equal(optimum, strat):
    iter_count+=1
    strat = np.copy(optimum)
    if P2 and P3:
       optimum = betterStrategy(S, U, all_rs, all_Qs, strat, P1, P2, P3)
    else:
      optimum = betterStrategy(S, U, all_rs, all_Qs, strat, P1)
  if P2 and P3:
    pi = getErgodicDist(S, U, optimum, P1, P2, P3)
  else:
    pi = getErgodicDist(S, U, optimum, P1)
  v = get_v(all_Qs, all_rs, pi, optimum)
  R = v[0]
  return init_strategy, optimum, iter_count, R
```

```
def betterStrategy(S, U, r, Q, strategy, P1, P2=None, P3=None):
  num_states = len(S)
  temp = np.copy(strategy)
  if P2 and P3:
    pi = getErgodicDist(S, U, temp, P1, P2, P3)
  else:
    pi = getErgodicDist(S, U, temp, P1)
  v = get_v(Q, r, pi, temp)
  cond1=[]
  for y in range(num_states):
    cond1_1 = [u for u in range(len(U)) if sum([Q[u][y][x]*v[0]
                                                for x in range(num_states)])==v[0]]
    cond1.append(cond1_1)
  cond2_right = [v[0] + v[y+1] for y in range(num_states)]
  cond2_left = [[r[y][u] + sum([Q[u][y][x]*v[x+1] for x in range(num_states)])
  for u in range(len(U))] for y in range(num_states)]
  for y in range(num_states):
    for u in range(len(U)-1, -1, -1):
      if cond2_left[y][u]<cond2_right[y] and any([cond1[y][i]==u
                                                  for i in range(len(cond1[y]))]):
        temp[y] = u
  return temp
```