Київський національний університет
імені Тараса Шевченка
Факультет комп’ютерних наук та кібернетики
Кафедра обчислювальної математики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 113 Прикладна математика

на тему:

Алгоритми розв’язання варіаційної нерівності

Виконав студент 4-го курсу
Скибицький Нікіта Максимович ________________

Науковий керівник:
доктор фіз.-мат. наук, професор
Семенов Володимир Вікторович ________________

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент ________________

Роботу розглянуто й допущено до захисту на засіданні кафедри обчислювальної математики
«___» __________________ 202_ р.,
протокол № ___
Завідувач кафедри
С. І. Ляшко ________________

Київ — 2020

РЕФЕРАТ

Обсяг роботи ?? сторінок, 8 ілюстрацій, 16 таблиць, ?? джерел посилань.

КЛЮЧОВІ СЛОВА.

Об’єктом роботи є ?? Предметом роботи є ??

Метою роботи є ??

Методи розроблення: ?? Інструменти розроблення: ??

Результати роботи: ??

??

ЗМІСТ

1 Вступ

1.1 Варіаційна нерівність

Розглянемо абстрактний (тобто поки що не накладаємо на нього ніяких обмежень і не вимагаємо від нього ніяких властивостей) оператор A який діє на підмножині C гільбертового простору H.

Визначення (варіаційної нерівності)**.** Кажуть, що для точки $x \in C$ виконується *варіаційна нерівність* якщо

$$\langle A(x), y - x \rangle \geq 0, \quad \forall y \in C. \tag{1.1}$$

Твердження 1. У випадку $C = H$ виконання варіаційної нерівності для точки x рівносильне виконанню рівності $A(x) = 0$.

Доведення. Справді, у випадку $C = H$ точка y пробігає увесь простір H. Тому для довільної фіксованої точки x точка $y - x$ також пробігає увесь простір H. Візьмемо y такий, що $y - x = -A(x)$, тоді

$$\langle A(x), y - x \rangle = \langle A(x), -A(x) \rangle = - \|A(x)\|^2 \leq 0, \tag{1.2}$$

причому рівність можлива лише якщо $A(x) = 0$. Отже, варіаційна нерівність може виконуватися тоді і тільки тоді, коли $A(x) = 0$. $\square$

1.2 Зв’язок із задачами оптимізації

Прояснимо зв’язок варіаційної нерівності із задачами оптимізації.

Твердження 2. Для задачі

$$f \to \min_C \tag{1.3}$$

у випадку опуклості як f так і C критерієм того, що точка x є розв’язком є виконання нерівності

$$\langle f'(x), y - x \rangle \geq 0, \quad \forall y \in C. \tag{1.4}$$

Доведення. Запишемо лінійну апроксимацію f:

$$f(y) = f(x) + \langle f'(x), y - x \rangle + o\left(\|y - x\|\right). \tag{1.5}$$

Припустимо тепер, що другий доданок менше нуля для якогось $y = x + z$, тоді

$$f(x + z) - f(x) = \langle f'(x), z \rangle + o\left(\|z\|\right). \tag{1.6}$$

Розглянемо (з опуклості C випливає, що $x + \varepsilon z \in C$, а отже можемо підставляти таке y) тепер $y = x + \varepsilon z$, отримаємо

$$f(x + \varepsilon z) - f(x) = \langle f'(x), \varepsilon z \rangle + o\left(\|\varepsilon z\|\right). \tag{1.7}$$

З визначення $o(\cdot)$ зрозуміло, що при $\varepsilon \to +0$ знак правої частини визначає перший доданок.

Тобто, права частина буде від’ємоною для якогось достатньо малого ε. Але тоді від’ємною буде і ліва частина, $f(x + \varepsilon z) - f(x) < 0$. Але це означає, що $f(x + \varepsilon z) < f(x)$. Отже, x не є точкою мінімуму f на C. Отримане протиріччя завершує доведення. □

Зауваження. У випадку відсутності опуклості або f або C або і того і того, цей критерій перетворюється на необхідну умову.

1.3 Зв'язок із сідловими точками

Розглянемо тепер оптимізацію з обмеженнями, тобто задачу

$$f(x) \xrightarrow[\substack{g_i(x)\le 0\\ i=1\ldots n}]{} \min\,. \tag{1.8}$$

Для цієї задачі можна побудувати функцію Лагранжа,

$$L(x,y) = f + \sum_{i=1}^{n} y_i g_i(x), \tag{1.9}$$

де y_i — множники Лагранжа.

Постає задача пошуку сідлової точки (справді, якщо у f мінімум в $\bar{x}$, то у L в $(\bar{x},\bar{y})$ буде мінімум по x і максимум по y, і навпаки) функції L.

Визначення (сідлової точки)**.** Точка $(\bar{x},\bar{y})$ називається *сідловою точкою* функції L якщо

$$L(\bar{x},y) \le L(\bar{x},\bar{y}) \le L(x,\bar{y}) \quad \forall x\, \forall y \tag{1.10}$$

тобто по x маємо мінімум в $\bar{x}$, а по y — максимум в $\bar{y}$.

Можемо записати ці умови наступним чином:

$$\begin{cases} \langle \nabla_1 L(\bar{x},\bar{y}), x-\bar{x}\rangle \ge 0 \quad \forall x \in C_1 \subseteq H_1,\\ \langle -\nabla_2 L(\bar{x},\bar{y}), y-\bar{y}\rangle \ge 0 \quad \forall y \in C_2 \subseteq H_2, \end{cases} \tag{1.11}$$

Зауваження. Ці нерівності можна об'єднати в одну:

$$\langle \nabla_1 L(\bar{x},y), x-\bar{x}\rangle + \langle -\nabla_2 L(\bar{x},\bar{y}), y-\bar{y}\rangle \ge 0. \tag{1.12}$$

1.4 Ерроу та Гурвіц

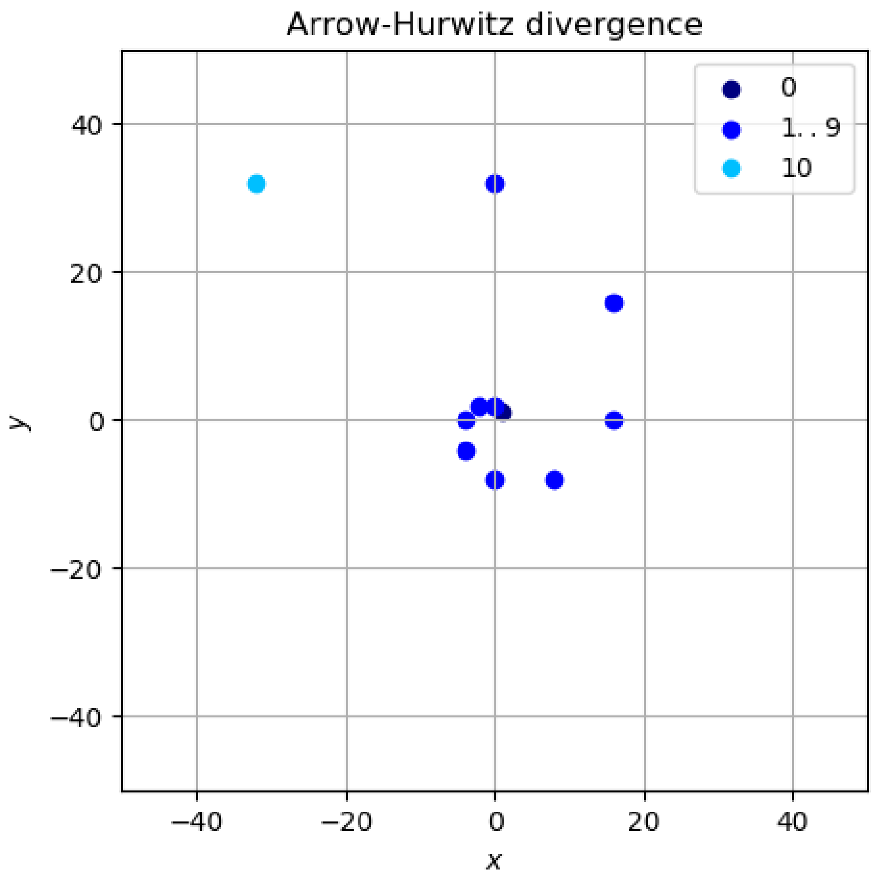
Приклад. Розглянемо тепер цілком конкретну функцію $L(x, y) = x \cdot y$ і спробуємо знайти її сідлову точку.

Розв'язок. Розглянемо алгоритм

$$\begin{aligned} x_{k+1} &:= x_k - \rho_k \nabla_1 L(x_k, y_k) = x_k - \rho_k y_k, \\ y_{k+1} &:= y_k + \rho_k \nabla_2 L(x_k, y_k) = y_k + \rho_k x_k, \end{aligned} \tag{1.13}$$

який називається *методом Ерроу-Гурвіца*.

Покладемо $(x_0, y_0) = (1, 1)$, $\rho_k \equiv 1$ і побачимо наступні ітерації:



Вони, очевидно, розбігаються, хоча здавалося що ми рухаємося у напрямку правильних градєнтів по кожній з компонент.

Зауваження. Для цієї задачі змінний розмір кроку нас не врятує, його зменшення тільки згладить спіраль по якій точки розбігаються.

Окрім емпіричних спостережень, ми можемо явно довести розбіжність, розглянемо для цього $|x_{k+1}|^2 + |y_{k+1}|^2$:

$$\begin{aligned} |x_{k+1}|^2 + |y_{k+1}|^2 = |x_k - \rho_k y_k|^2 + |y_k + \rho_k x_k|^2 = \\ = |x_k|^2 + \rho_k^2 \left(|x_k|^2 + |y_k|^2\right) + |y_k|^2 > |x_k|^2 + |y_k|^2, \end{aligned} \tag{1.14}$$

у той час як сідловою точкою, очевидно, є $(0, 0)$. □

Зауваження. Не зважаючи на розбіжність методу Ерроу-Гурвіца на такій простій задачі, Ерроу свого часу був удостоєний Нобелівськиї премії з економіки, за задачі які цим методом можна розв'язати.

Виникає закономірне зпитання а що ж це за задачі такі.

1.5 Сильно опуклі функції

Для відповіді на це питання нам доведеться ввести

Визначення (сильно опуклої функції)**.** Функція $f = f(x)$ називається μ*-сильно опуклою* для деякого $\mu > 0$ якщо

$$f(\alpha \cdot x + (1-\alpha) \cdot y) \leq \alpha \cdot f(x) + (1-\alpha) \cdot f(y) - \mu \cdot \alpha \cdot (1-\alpha) \cdot \|x-y\|^2 . \quad (1.15)$$

Про всяк введемо також трохи більш узагальнене

Визначення (сильно опуклої функції)**.** Функція $f = f(x)$ називається g*-сильно опуклою* для деякої g якщо

$$f(\alpha \cdot x + (1-\alpha) \cdot y) \leq \alpha \cdot f(x) + (1-\alpha) \cdot f(y) - \alpha \cdot (1-\alpha) \cdot g(\|x-y\|). \quad (1.16)$$

Можемо записати також альтернетивне визначення μ-сильно опуклості:

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \mu \cdot \|x-y\|^2 . \quad (1.17)$$

Так от виявляється, що для задач із сильно опуклою функцією f метод Ерроу-Гурвіца збігається. Доведення наводиться нижче у більш загальному випадку, а поки що останннє

Зауваження. Існують певні ергодичні теореми, які стверджують збіжність середнього значення

$$(\tilde{x}_n, \tilde{y}_n) = \left(\frac{x_0 + \ldots + x_n}{n+1}, \frac{y_0 + \ldots + y_n}{n+1} \right) \quad (1.18)$$

до якоїсь сідлової точки $(\bar{x}, \bar{y})$, але подібні усереднені методи не є практичними, адже вони збігаються дуже повільно, а у задачі вище вже за тисячу ітерацій числа стають настільки великі що машинні похибки “переважають” усю теорію.

1.6 Монотонні оператори

Нагадаємо, що ми намгаємося знайти точку $x \in C$ яка задовольняє варіаційній нерівності

$$\langle Ax, y - x \rangle \geq 0 \quad \forall y \in C, \tag{1.19}$$

де оператор A, взагалі кажучи, не нерозтягуючий.

Подивимося на цю задачу як на задачу знаходження нерухомої точки оператора

$$T : x \mapsto P_C\left(x - \rho Ax\right), \tag{1.20}$$

де $\rho > 0$. Одразу зауважимо, що ці міркування приводять нас до наступного алгоритму

$$x_{k+1} \coloneqq P_C\left(x_k - \rho_k Ax_k\right), \tag{1.21}$$

збіжність якого ми зараз і проаналізуємо.

Взагалі хотілося б (відомо багато теорем щодо збіжності описаного алгоритму за таких умов) щоб оператор T був нерозтягуючим. Маємо:

$$\begin{aligned} \|Tx - Ty\|^2 &\leq \|x - y - \rho(Ax - Ay)\|^2 \leq \\ &\leq \|x - y\|^2 - 2\rho\left\langle Ax - Ay, x - y\right\rangle + \rho^2\left\|Ax - Ay\right\|^2. \end{aligned} \tag{1.22}$$

Для подальших оцінок нам знадобиться поняття монотонного оператора.

Визначення (монотонного оператора)**.** Оператор A: $H \to H$ називається *монотонним* якщо

$$\langle Ax - Ay, x - y \rangle \geq 0 \quad \forall x\, \forall y. \tag{1.23}$$

Поняття монотонності для операторів відіграє схожу роль з поняттям монотонності функцій.

Приклад. Оператор A називається *опуклим* якщо його градієнт ∇A монотонний.

Аналогічно до μ-сильно опуклих функцій існують μ-сильно опуклі оператори, для визначення яких вводиться

Визначення (сильно монотонного оператора)**.** Оператор A: $H \to H$ називається *μ-сильно монотонним* зі сталою $\mu > 0$ якщо

$$\langle Ax - Ay, x - y \rangle \geq \mu \cdot \|x - y\|^2 \quad \forall x\, \forall y. \tag{1.24}$$

Якщо оператор A — μ-сильно монотонний то можемо продовжити ланцюжок оцінок:

$$\|x-y\|^2 - 2\rho\left\langle Ax - Ay, x-y\right\rangle + \rho^2\|Ax-Ay\|^2 \leq \\ \leq \|x-y\|^2 - 2\rho\mu\|x-y\|^2 + \rho^2\|Ax-Ay\|^2 . \quad (1.25)$$

Якщо ж при цьому оператор A ще й L-ліпшицеви (ліпшицевий з константою L, тобто $\|Ax-Ay\| \leq L\cdot\|x-y\|$), то можемо продовжити ще:

$$\|x-y\|^2 - 2\rho\mu\|x-y\|^2 + \rho^2\|Ax-Ay\|^2 \leq \\ \leq \|x-y\|^2 - 2\rho\mu\|x-y\|^2 + \rho^2L^2\|x-y\|^2 = \\ = (1-2\rho\mu+\rho^2L^2)\|x-y\|^2 = \kappa(\rho)\cdot\|x-y\|^2 . \quad (1.26)$$

тобто достатньо обрати ρ так, щоб $\kappa(\rho) \in (0,1)$.

Розв'язуючи отриману квадратну нерівність знаходимо:

$$\rho \in \left(0, \frac{2\mu}{L^2}\right), \quad (1.27)$$

тобто знайшли цілий інтервал значень ρ для яких наш алгоритм буде збіжним.

Здавалося б все добре, але подивимося, у якій точці досягається мінімум $\kappa(\rho)$:

$$\tilde{\rho} = \frac{\mu}{L^2}, \quad (1.28)$$

і чому він дорівнює:

$$\kappa(\tilde{\rho}) = 1 - 2\rho\mu + \rho^2L^2 = 1 - 2\frac{\mu}{L^2}\mu + \left(\frac{\mu}{L^2}\right)^2 L^2 = 1 - \frac{\mu^2}{L^2}. \quad (1.29)$$

Зауваження. На жаль, правда життя така, що μ зазвича доволі мале, а L навпаки — доволі велике, тому $\rho < 1$ але зовсім трохи. А це у свою чергу означає повільну збіжність.

1.7 Регуляризація

У той же час майже довільну опуклу функцію f можна замінити (цей процес називається регуляризацією) на ε-сильно опуклу функцію $f_\varepsilon = f + \varepsilon \|x\|^2$, тому може здатися, що всі наші проблеми розв'язані.

Так, у загальному випадку для монотонного оператора A можна розглянути оператор $A_\varepsilon = A + \varepsilon \mathbf{1}$, де $\mathbf{1}$, $x \mapsto x$ — *одиничний* (*тотожний*) *оператор*. Тоді можемо записати

$$\langle A_\varepsilon x - A_\varepsilon y, x - y \rangle = \underbrace{\langle Ax - Ay, x - y \rangle}_{\geq 0} + \varepsilon \cdot \|x - y\|^2 \geq \varepsilon \cdot \|x - y\|^2 , \tag{1.30}$$

тобто оператор A_ε є ε-сильно опуклим.

Це наштовхує на думки про побудову алгоритму з ітераціями вигляду

$$x_{k+1} \coloneqq P_C \left(x_k - \rho_k A_{\varepsilon_k} x_k \right) , \tag{1.31}$$

але тоді постає ще ряд запитань, наприклад які умови мають задовольняти $\{\rho_k\}_{k=1}^\infty$ і $\{\varepsilon_k\}_{k=1}^\infty$ для збіжності цього алгоритму. Поки що ці запитання лишаємо без відповіді.

1.8 Зв'язок із мережевими іграми і рівновагою Неша

Цей розділ взято з доповіді [Asu Özdaglar, 2018].

У багатьох соціальних та економіних задачах, рішення окремих індивідів (агентів) залежать лише від дій їхніх друзів, колег, однолітків чи суперників. Як приклади можна навести:

- Поширення інновацій, стилю життя.
- Формування громадських думок і соціальне навчання.
- Суперництво між конкурентними фірмами.
- Забезпечення суспільних благ.

Такі взаємодії можна промоделювати мережевою грою, що означає виконання наступних припущень:

- Агенти взаємодіють по ребрам мережі, представленої графом.
- Виграш кожного гравця залежить від його власних дій і від *агрегованого* значення дій агентів у його околі.

Формальніше, модель мережевої гри наступна: n агентів взаємодіють по мережі $G \in \mathbb{R}^{n\times n}$:

$$\begin{cases} G_{i,j} > 0 & \text{вплив } j \text{ на } i, \\ G_{i,i} = 0 & \text{без петель.} \end{cases} \tag{1.32}$$

У кожного агента i є:

- стретегія $x^i \in \mathcal{X}^i$, де $\mathcal{X}^i \subset \mathbb{R}^n$ — допустима множина стратегій ;
- цільова функція $J^i(x^i, z^i(x)) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, де $z^i(x) := \sum_{j=1}^n G_{i,j} x^j$ — агрегатор.

Кожен агент намагється навчитися обчислювати свою оптимальну відповідь:

$$x^i_{\text{br}}(z^i) := \operatorname*{argmin}_{x^i \in \mathcal{X}^i} J^i(x^i, z^i). \tag{1.33}$$

Нагадаємо

Визначення (рівноваги за Нешем)**.** Множина стратегій $\{\bar{x}_i\}_{i=1}^n$ називається *рівновагою за Нешем* якщо для кожного гравця i, $\bar{x}^i \in \mathcal{X}^i$:

$$J^i\left(\bar{x}^i, \sum_{j=1}^n G_{i,j}\bar{x}^j\right) \le J^i\left(x^i, \sum_{j=1}^m G_{i,j}\bar{x}^i\right), \quad \forall x^i \in \mathcal{X}^i. \tag{1.34}$$

Можна показати, що при виконанні наступних припущень:

- $\mathcal{X}^i \subset \mathbb{R}^n$ — замкнені, опуклі та обмежені;
- $J^i(x^i, z^i(x))$ опукла по x^i, для кожного вектора доповнюючих стратегій $x^{-i} \in \mathcal{X}^{-i}$;
- $J^i(x^i, z^i) \in C^2$ по x^i і z^i.

справджується наступне

Твердження 3. $\bar{x}$ є рівновагою за Нешем $\iff$ $\bar{x}$ є розв'язком варіаційної нерівності із допустимою множиною $\mathcal{X}$ і функцією F визначеними наступним чином:

$$\mathcal{X} \coloneqq \mathcal{X}^1 \times \cdots \times \mathcal{X}^n; \tag{1.35}$$

$$F(x) \coloneqq [F^i(x)]_{i=1}^n \coloneqq \begin{bmatrix} \nabla_{x^1} J^1(x^1, z^1(x)) \\ \vdots \\ \nabla_{x^n} J^n(x^n, z^n(x)) \end{bmatrix} \tag{1.36}$$

Твердження 4 (Facchinei та Pang, 2003)**.** Якщо окрім цього, якобіан гри F строго монотонний, то рівновага за Нешем існує та єдина.

1.9 Зв'язок із транспортними мережами

Цей розділ взято з книги А. В. Гасникова, «Введение в математическое моделирование транспортных потоков».

Розглянемо один із підходів до моделювання і дослідження транспортних потоків, що базується на теорії конкурентної безкоаліційної рівноваги, яка дозволяє описати доволі адекватних механізм функціонування автомобільних вулично-дорожніх мереж (ВДМ).

Моделі, яки ми розглянемо, застосовуються для отримання прогнозних оцінок завантаженності елементів транспортної мережі. Подібні задачі цікаві, зокрема, тим, що є одним із інструкментів об'єктиної оцінки ефективності проектів з модифікації ВДМ з точки зору розвантаження найбільш завантажених і проблемних ділянок доріг.

Моделювання транспортних потоків як задача прийняття рішень

Для визначення обсягів завантаження ВДМ у першу чергу необхідно з'ясувати, чим керуються водії, обираючи той чи інший маршрут. Поведінкові принципи користувачів транспортної мережі були остаточно сформульовані Вардропом у [J. Wardrop, Some Theoretical Aspects of Road Traffic Research, 1952], де він наводить дві можливі ситуації:

1. користувачі мережі незалежно один від одного обирають маршрути з мінімальними затратами; (user optimization)
2. користувачі мережі обирають маршрути з мінімальними сумарними затратами. (system optimization)

Визначення. Ці принципи називаються, відповідно, *першим і другим принципами Вардропа*.

Перший принцип Вардропа відповідає конкурентнії безколіаційній рівновазі (жоден окремий користувач не може знизити свої витрати відхилившись від свого поточного маршруту). Цей принцип передбачає ідеальну інформованість окремих користувачів про витрати на усіх шляхах (сьогодні це досягається за допомогою розумних навігаторів), а також відсутність суттєвого впливу на витрати на шляхах зі сторони одного конкретного користувача (не виконується для вантажних автомобілів а також аварійних ситуацій, але здебільшого виконується). Резюмуючи, перший принцип Вардропа сьогодні доволі точно описує поведінкові принципи користувачів ВДМ, а не є якоюсь надміру ідеалізованою абстракцією.

Формалізація задачі

Виходячи з наведених міркувань, побудуємо економіко-математичну моделі розподілу транспортних потоків у ВДМ. Транспортну мережу опишему у вигляді орієнтованого графу $G(V, E)$, де V — множина вершин (перехресть), E — множина орієнтованих дуг мережі (доріг між сусідніми перехрестями).

При дослідженні потокоформуючих факторів у множині вершин розглянемо дві підмножини. Перше, $S \subset V$, містить пункти, що породжують потоки; елементи множини S (sources) назвемо джерелами/витоками. Друге, $D \subset V$, містить пункти, що поглинають потоки; елементи множини D (destinations) назвемо стоками. Для задачі моделюванні трудових міграцій (для ранкових годин пік), джерелами будуть спальні райони міста, а стоками — ділові райони міста (для вечірніх годин ситуація рівно протилежна).

Через $P_{s,d}$ будемо позначати множину шляхів з витоку s у стік d. Окремі шляхи із витоку s у стік d будемо позначати $p_{s,d}$, або просто p якщо витік і стік несуттєві. Через f_p позначимо потік на шляху p, через $c_p(F)$ — маргинальні затрати на шляху p якщо загальна матриця потоків на шляхах дорівнює F (природніми прикладами колм затрати на одному шляху залежать від завантаженості інших є перехрестя головних і другорядних доріг, або ж паралельні дороги). Зрозуміло, що умова рівноваги має наступний вигляд:

$$\text{якщо } f_{p_{s,d}} > 0, \text{ то } c_p(F) = \min_{p' \in P_{s,d}} c_{p'}(F). \tag{1.37}$$

Окрім цього, для кожної пари (s, d) існує певний сталий (у випадку трудових міграцій) попит на транспорт, який будемо позначати $\rho_{s,d}$. Зрозуміло, що оптимальний рівноважний потік також має задовольняти наступним консервативним умовам:

$$\sum_{p \in P(s,d)} f_p = \rho_{s,d}. \tag{1.38}$$

Твердження 5. За таких умов допустима множина транспортних потоків $\mathcal{F}$ володіє гарними властивостями, а саме вона замкнута, опукла, і непорожня.

Зведення до варіаційної нерівності

Якщо залежність транспортних витрати від обсягів завантаження ВДМ є монотонною і неперевною, то пошук рівноважних потоків може бути зведений до розв'язання варіаційної нерівності.

Теорема 1. Потік $F^\star \in \mathcal{F}$ задовольняє умову рівноваги (1.37) тоді і лише тоді, коли він є розв'язком варіаційної нерівності

$$\langle C(F^\star), F - F^\star \rangle \geq 0, \quad \forall F \in \mathcal{F}, \tag{1.39}$$

де у матрицю C зібрані маргинальні транспортні витрати на усіх шляхах.

1.10 Подальші припущення

Надалі будемо розв'язувати наступну задачу:

$$\text{знайти } x \in C : \quad \langle Ax, y - x \rangle \geq 0, \quad \forall y \in C. \tag{1.40}$$

Також будемо вважати, що виконані наступні умови:

- множина $C \subseteq H$ — опукла і замкнена;
- оператор $A : H \to H$ — монотонний і ліпшицевий (із константою L);
- множина розв'язків (1.40) непорожня.

2 Алгоритми

2.1 Класичні алгоритми

Серед численних алгоритмів розв'язування (1.40) розглянемо три базових:

Алгоритм 1 (Корпелевич)**. Ініціалізація.** Вибираємо елементи x_1, $\lambda \in \left(0, \frac{1}{L}\right)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{2.1}$$

Якщо $x_n = y_n$ то зупиняємося і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A y_n), \tag{2.2}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Алгоритм 2 (P. Tseng)**. Ініціалізація.** Вибираємо елементи x_1, $\lambda \in \left(0, \frac{1}{L}\right)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{2.3}$$

Якщо $x_n = y_n$ то зупиняємося і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$x_{n+1} = y_n - \lambda(A y_n - A x_n), \tag{2.4}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Алгоритм 3 (Попов)**. Ініціалізація.** Вибираємо елементи x_1, y_0, $\lambda \in \left(0, \frac{1}{3L}\right)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A y_{n-1}). \tag{2.5}$$

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A y_n). \tag{2.6}$$

Якщо $x_{n+1} = x_n = y_n$ то зупиняємо алгоритм і x_n — розв'язок, інакше покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Зауваження. У наведеному вище вигляді алгоритми Tseng'а і Попова обчислюють оператор A тричі і двічі на кожну ітерацію відповідно. На цьому можна заощадити якщо кешувати обчислення оператора A. У випадку алгоритма Tseng'а спосіб кешування очевидний: один раз обчислюємо Ax_n і двічі використовуємо його (для y_n та x_{n+1}). У випадку алгоритма Попова кешування допомагає за рахунок того, що значення Ay_n використовується один раз на ітерації n для обчислення x_{n+1}, і ще раз на ітерації $n + 1$ для обчислення значення y_{n+1}.

В теорії, у випадку коли P_C обчислювати дешево (наприклад, коли це можливо аналітично), а A обчислювати дорого, такий трюк допомагає пришвидшити алгоритм Tseng'а у 1.5, а алгоритм Попова — у 2 рази.

2.2 Адаптивні алгоритми

Не так давно з'явилися адаптивні алгоритми, тобто такі, що не вимагають знання константи Ліпшиця. Наведемо адаптивні версії розгялнутих раніше алгоритмів:

Алгоритм 4 (Адаптивний Корпелевич)**. Ініціалізація.** Вибираємо елементи x_1, $\tau \in (0,1)$, $\lambda \in (0,+\infty)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{2.7}$$

Якщо $x_n = y_n$ то зупиняємося і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A y_n). \tag{2.8}$$

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } \langle Ax_n - Ay_n, x_{n+1} - y_n \rangle \leq 0, \\ \min\left\{ \lambda_n, \dfrac{\tau}{2} \dfrac{\|x_n - y_n\|^2 + \|x_{n+1} - y_n\|^2}{\langle Ax_n - Ay_n, x_{n+1} - y_n \rangle} \right\}, \text{ інакше.} \end{cases} \tag{2.9}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Зауваження. У алгоритмі 4 можна робити і так:

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ax_n - Ay_n = 0, \\ \min\left\{ \lambda_n, \tau \dfrac{\|x_n - y_n\|}{\|Ax_n - Ay_n\|} \right\}, \text{ інакше.} \end{cases} \tag{2.10}$$

Алгоритм 5 (Адаптивний Tseng)**. Ініціалізація.** Вибираємо елементи x_1, $\tau \in (0,1)$, $\lambda \in (0,+\infty)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{2.11}$$

Якщо $x_n = y_n$ то зупиняємося і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$x_{n+1} = y_n - \lambda(Ay_n - Ax_n), \tag{2.12}$$

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ax_n - Ay_n = 0, \\ \min\left\{\lambda_n, \tau\dfrac{\|x_n - y_n\|}{\|Ax_n - Ay_n\|}\right\}, \text{ інакше.} \end{cases} \tag{2.13}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Алгоритм 6 (Адаптивний Попов)**. Ініціалізація.** Вибираємо елементи x_1, y_0, $\tau \in (0, \frac{1}{3})$, $\lambda \in (0, +\infty)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda Ay_{n-1}). \tag{2.14}$$

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda Ay_n). \tag{2.15}$$

Якщо $x_{n+1} = x_n = y_n$ то зупиняємо алгоритм і x_n — розв’язок, інакше переходимо на

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } \langle Ay_{n-1} - Ay_n, x_{n+1} - y_n\rangle \le 0, \\ \min\left\{\lambda_n, \dfrac{\tau}{2}\dfrac{\|y_{n-1} - y_n\|^2 + \|x_{n+1} - y_n\|^2}{\langle Ay_{n-1} - Ay_n, x_{n+1} - y_n\rangle}\right\}, \text{ інакше.} \end{cases} \tag{2.16}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Зауваження. У алгоритмі 6 можна робити і так:

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ay_{n-1} - Ay_n = 0, \\ \min\left\{\lambda_n, \tau\dfrac{\|y_{n-1} - y_n\|}{\|Ay_{n-1} - Ay_n\|}\right\}, \text{ інакше.} \end{cases} \tag{2.17}$$

2.3 Алгоритм Маліцького—Там'а

Зовсім нещодавно (у 2015-ому році) Юра Маліцький запропонував наступну схему:

Алгоритм 7 (Маліцький—Там)**. Ініціалізація.** Вибираємо елементи $x_1, x_0 \in H$, $\lambda \in (0, \frac{1}{2L})$. Покладаємо $n = 1$.

Крок. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A x_n - \lambda(Ax_n - Ax_{n-1})). \tag{2.18}$$

Якщо $x_{n+1} = x_n = x_{n-1}$ то зупиняємо алгоритм і x_n — розв'язок, інакше покладаємо $n \coloneqq n + 1$, і повторюємо

Алгоритм 8 (Адаптивний Маліцький—Там)**. Ініціалізація.** Вибираємо елементи $x_1, x_0 \in H$, $\lambda_1, \lambda_0 > 0$, $\tau \in (0, \frac{1}{2})$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A x_n - \lambda(Ax_n - Ax_{n-1})). \tag{2.19}$$

Якщо $x_{n+1} = x_n = x_{n-1}$ то зупиняємо алгоритм і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$\lambda_{n+1} = \min\left\{\lambda_n, \tau\frac{\|x_{n+1} - x_n\|}{\|Ax_{n+1} - Ax_n}\right\}, \tag{2.20}$$

покладаємо $n \coloneqq n + 1$ і переходимо на **Крок 1.**

3 Задачі

Для порівняння алгоритмів нам знадобляться тестові задачі різної складності та різних розмірів. У якості таких задачі розглянемо:

3.1 Перша задача

Класичний приклад. Допустимою множиною є увесь простір: $C = \mathbb{R}^m$, а $F(x) = Ax$, де A — квадратна $m \times m$ матриця, елементи якої визначаються наступним чином:

$$a_{i,j} = \begin{cases} -1, & j = m - 1 - i > i, \\ 1, & j = m - 1 - i < i, \\ 0, & \text{інакше}. \end{cases} \tag{3.1}$$

Зауваження. Тут і надалі нуметрація рядків/стовпчиків матриць, а також елементів масивів починається з нуля. Якщо у вашій мові програмування нумерація починається з одиниці то у виразах вище замість $m - 1$ має бути $m + 1$.

Це визначає матрицю, чия бічна діагональ складається з половини одиниць і половини мінус одиниць, а решта елементів якої нульові. Для наглядності наведемо декілька преших матриць, для $m = 2, 4, 6$:

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.2}$$

Для парних m нульовий вектор є розв'язком відповідної варіаційної нерівності (1.40).

Зауваження. Для цієї задачі $P_C = \mathrm{Id}$, а тому алгоритми Корпелевич і Tseng'а еквівалентні. Втім, некешована версія алгоритму Tseng'а буде працювати повільніше, що ми скоро і побачимо.

3.2 Друга задача

Візьмемо $F(x) = Mx + q$, де матриця M генерується наступний чином:

$$M = AA^{\mathsf{T}} + B + D, \tag{3.3}$$

де всі елементи $m \times m$ матриці A і $m \times m$ кососиметричної матриці B обираються рівномірно випадково з $(-5, 5)$, а усі елементи діагональної матриці D вибираються рівномірно випадково з $(0, 0.3)$ (як наслідок, матриця M додатно визначена), а кожен елемент q обирається рівномірно випадково з $(-500, 0)$. Допустимою множиною є

$$C = \{x \in \mathbb{R}^m_+ | x_1 + x_2 + \cdots + x_m = m\}, \tag{3.4}$$

а за початкове наближення береться $x_1 = (1, \ldots, 1)$. Для цієї задачі $L = \|M\|$, $\varepsilon = 10^{-3}$.

Допустима множина цієї задачі — так званий *probability symplex* (з точністю до константи m). Для проектування $\vec{y}$ на нього ми використовували наступний явний

Алгоритм 9.

Крок 1. Відсортувати елементи $\vec{y}$ і зберегти в $\vec{u}$: $u_1 \geq \cdots \geq u_m$.

Крок 2. Знайти $k = \max j$: $u_j + \frac{1}{j}\left(m - \sum_{i=1}^{j} u_i\right) > 0$.

Крок 3. Видати вектор з елементами $x_i = \max\{y_i + \lambda, 0\}$, $\lambda = \frac{1}{k}\left(m - \sum_{i=1}^{k} u_i\right)$.

Цей алгоритм взято із статті [J. Duchi, Sh. Shalev-Shwartz, Y. Singer, T. Chandra, 2008].

3.3 Четверта задача

$$
\begin{aligned}
F(x) &= F_1(x) + F_2(x), \\
F_1(x) &= (f_1(x), f_2(x), \ldots, f_m(x)), \\
F_2(x) &= Dx + c, \\
f_i(x) &= x_{i-1}^2 + x_i^2 + x_{i-1}x_i + x_i x_{i+1}, \quad m = 1, 2, \ldots, m, \\
x_0 &= x_{m+1} = 0,
\end{aligned} \tag{3.5}
$$

де D — квадратна $m \times m$ матриця з наступними елементами:

$$
d_{i,j} = \begin{cases} 1, & j = i-1, \\ 4, & j = i, \\ -2, & j = i+1, \\ 0, & \text{інакше}, \end{cases} \tag{3.6}
$$

$c = (-1, -1, \ldots, -1)$. Допустимою множиною є $C = \mathbb{R}^m_+$, а початкова точка $x_1 = (0, 0, \ldots, 0)$.

Для кращого розуміння наведемо матрицю D для кількох перших $m = 3, 4, 5$:

$$
\begin{pmatrix} 4 & -2 & 0 \\ 1 & 4 & -2 \\ 0 & 1 & 4 \end{pmatrix} \qquad
\begin{pmatrix} 4 & -2 & 0 & 0 \\ 1 & 4 & -2 & 0 \\ 0 & 1 & 4 & -2 \\ 0 & 0 & 1 & 4 \end{pmatrix} \qquad
\begin{pmatrix} 4 & -2 & 0 & 0 & 0 \\ 1 & 4 & -2 & 0 & 0 \\ 0 & 1 & 4 & -2 & 0 \\ 0 & 0 & 1 & 4 & -2 \\ 0 & 0 & 0 & 1 & 4 \end{pmatrix} \tag{3.7}
$$

Зауваження. Матриця тридіагональна, ми цим скористаємося.

4 Результати

Тестування відбувалося на машині із процесором Intel Core i7-8550U 1.99GHz під 64-бітною версією операційної системи Windows.

4.1 Перша задача, неадаптивні алгоритми

Для усіх алгоритмів у якості початкового наближення ми брали $x_1 = (1, \ldots, 1)$, $\varepsilon = 10^{-6}$, $\lambda = 0.4$ (константа Ліпшиця цієї задачі дорівнює одиниці: $L = 1$).

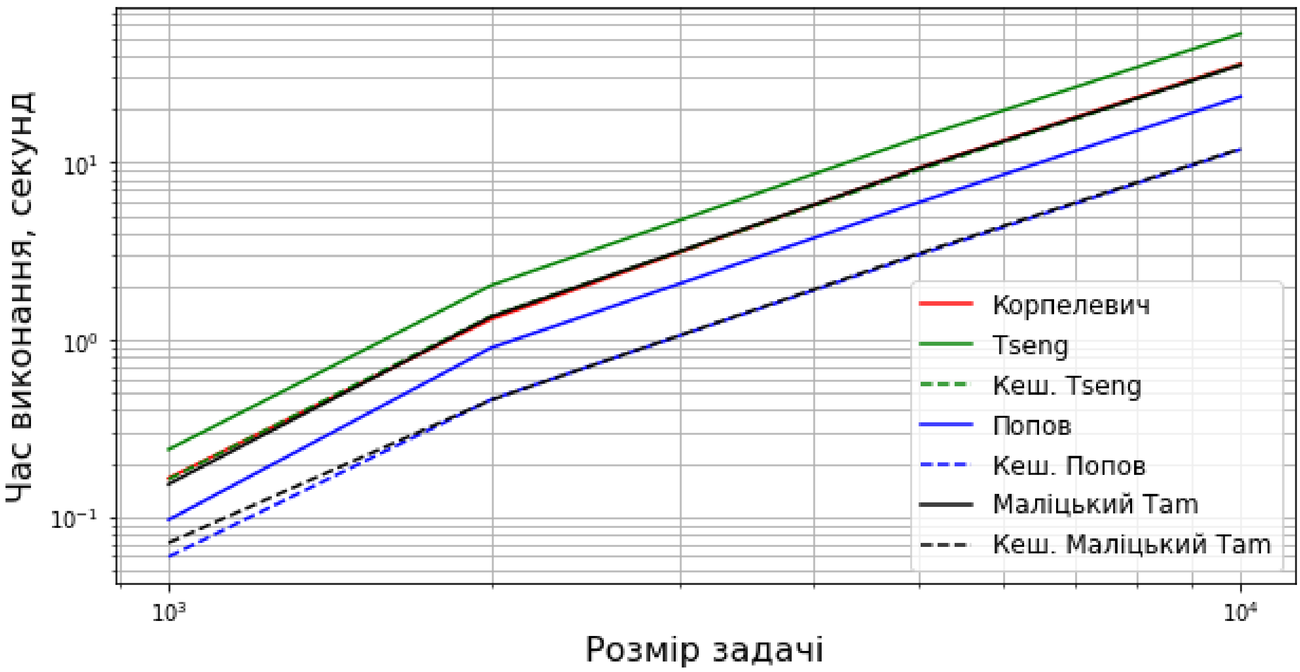


Рис. 4.1: Результати неадаптивних алгоритмів на першій задачі

І справді, бачимо що алгоритм Корпелевич і кешований алгоритм Tseng'а справді показують майже однакові результати, а також обидві некешовані версії програють кешованим. Та сама інформація у табличці, для зручності:

Розмір задачі	1000	2000	5000	10000
Корпелевич	0.17	1.31	9.30	36.06
Tseng	0.24	2.03	13.75	53.33
Кеш. Tseng	0.16	1.36	9.06	35.61
Попов	0.10	0.90	5.94	23.53
Кеш. Попов	0.06	0.46	3.01	11.87
Маліцький Tam	0.15	1.35	9.26	35.48
Кеш. Маліцький Tam	0.07	0.46	3.06	11.96

Таблиця 4.1: Час виконання, секунд

Розмір задачі	1000	2000	5000	10000
Корпелевич	228	233	239	244
Tseng	228	233	239	244
Кеш. Tseng	228	233	239	244
Попов	151	154	158	161
Кеш. Попов	151	154	158	161
Маліцький Tam	153	156	160	163
Кеш. Маліцький Tam	153	156	160	163

Таблиця 4.2: Число ітерацій

4.2 Перша задача, адаптивні алгоритми

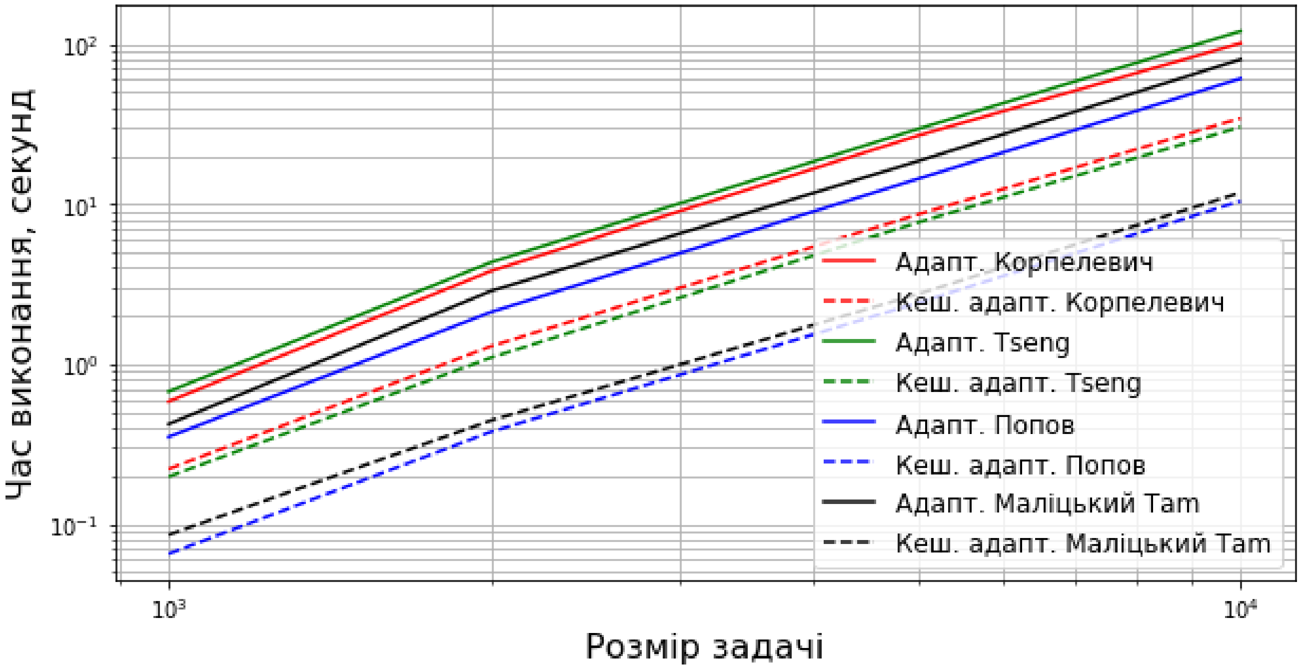


Рис. 4.2: Результати адаптивних алгоритмів на першій задачі

Та сама інформація у табличці, для зручності:

Розмір задачі	1000	2000	5000	10000
Адапт. Корпелевич	0.58	3.83	26.84	101.54
Кеш. адапт. Корпелевич	0.22	1.30	8.63	34.52
Адапт. Tseng	0.67	4.34	29.54	120.72
Кеш. адапт. Tseng	0.20	1.10	7.69	30.38
Адапт. Попов	0.35	2.12	14.40	61.09
Кеш. адапт. Попов	0.07	0.38	2.42	10.48
Адапт. Маліцький Tam	0.42	2.87	18.57	80.46
Кеш. адапт. Маліцький Tam	0.09	0.45	2.74	11.82

Таблиця 4.3: Час виконання, секунд

Алгоритма Попова програє своїй неадаптивній версії.

Щодо кількості ітерацій ситуація схожа:

Розмір задачі	1000	2000	5000	10000
Адапт. Корпелевич	229	234	240	245
Кеш. адапт. Корпелевич	229	234	240	245
Адапт. Tseng	229	234	240	245
Кеш. адапт. Tseng	197	202	208	213
Адапт. Попов	131	134	138	142
Кеш. адапт. Попов	131	134	138	142
Адапт. Маліцький Tam	154	157	161	164
Кеш. адапт. Маліцький Tam	154	157	161	164

Таблиця 4.4: Число ітерацій

4.3 Перша задача із розрідженими матрицями, неадаптивні алгоритми

Нескладно помітити, що матриця A дуже розріджена, що наводить на ідею скористатися модулем scipy.sparse для ефективної роботи з розрідженими матрицями. Це дозволить нам розв'язувати задачу для значно більших m.

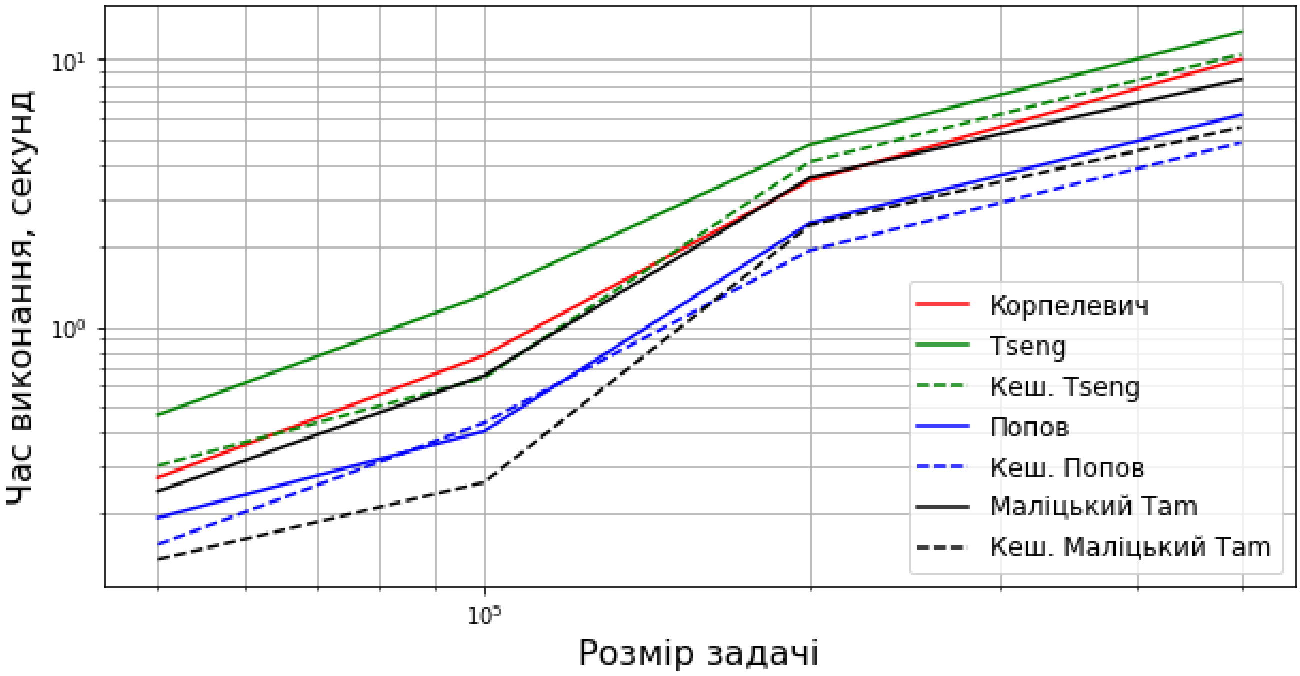


Рис. 4.3: Результати неадаптивних алгоритмів на першій задачі із розрідженими матрицями

Та сама інформація у табличці, для зручності:

Розмір задачі	50000	100000	200000	500000
Корпелевич	0.27	0.78	3.52	10.01
Tseng	0.47	1.31	4.80	12.69
Кеш. Tseng	0.30	0.64	4.14	10.45
Попов	0.19	0.41	2.44	6.20
Кеш. Попов	0.15	0.44	1.93	4.88
Маліцький Там	0.24	0.66	3.61	8.42
Кеш. Маліцький Там	0.13	0.26	2.40	5.57

Таблиця 4.5: Час виконання, секунд

Розмір задачі	50000	100000	200000	500000
Корпелевич	255	260	265	271
Tseng	255	260	265	271
Кеш. Tseng	255	260	265	271
Попов	168	171	174	178
Кеш. Попов	168	171	174	178
Маліцький Там	170	173	176	180
Кеш. Маліцький Там	170	173	176	180

Таблиця 4.6: Число ітерацій

Зауваження. Тут перевага кешування вже не така очевидна, адже ми значно здешевили обчислення оператора A, хоча все ще присутня.

4.4 Перша задача із розрідженими матрицями, адаптивні алгоритми

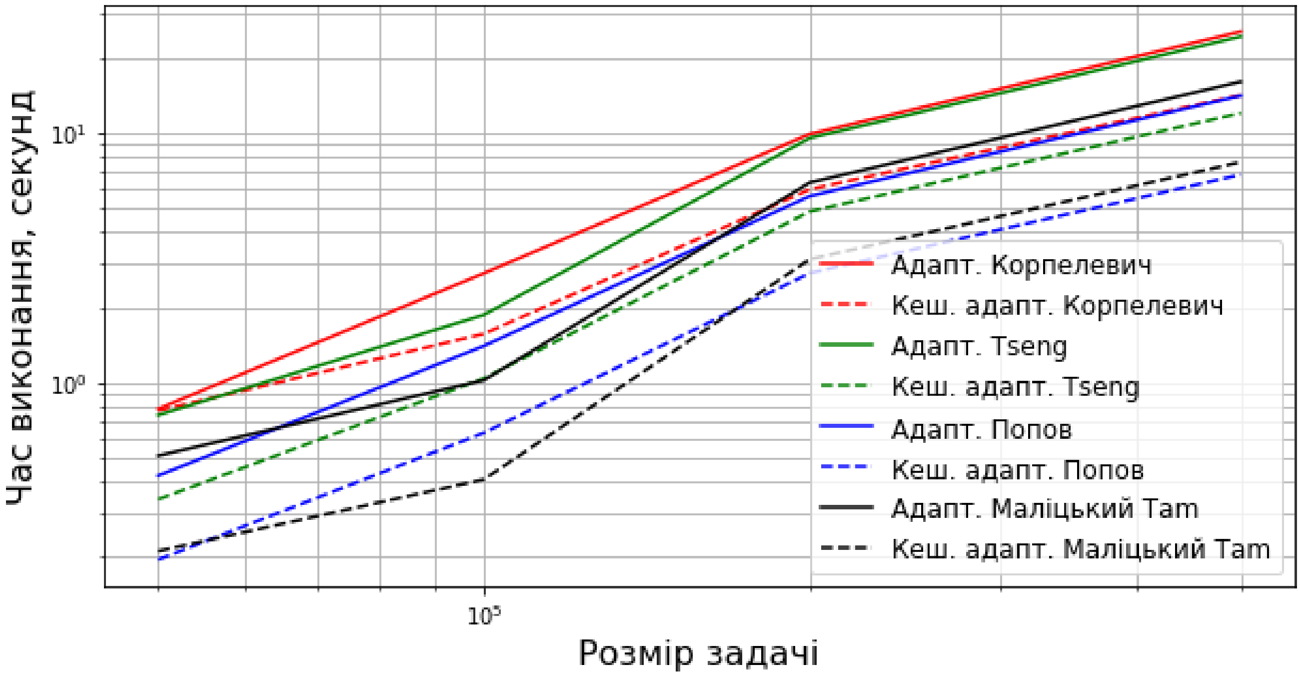


Рис. 4.4: Результати адаптивних алгоритмів на першій задачі із розрідженими матрицями

Та сама інформація у табличці:

Розмір задачі	50000	100000	200000	500000
Адапт. Корпелевич	0.78	2.74	9.91	25.52
Кеш. адапт. Корпелевич	0.77	1.57	5.95	14.19
Адапт. Tseng	0.74	1.86	9.55	24.37
Кеш. адапт. Tseng	0.34	1.04	4.85	12.03
Адапт. Попов	0.42	1.40	5.59	14.12
Кеш. адапт. Попов	0.19	0.63	2.74	6.81
Адапт. Маліцький Tam	0.51	1.02	6.33	16.06
Кеш. адапт. Маліцький Tam	0.21	0.41	3.12	7.64

Таблиця 4.7: Час виконання, секунд

Розмір задачі	50000	100000	200000	500000
Адапт. Корпелевич	256	261	266	272
Кеш. адапт. Корпелевич	256	261	266	272
Адапт. Tseng	256	261	266	272
Кеш. адапт. Tseng	224	229	234	240
Адапт. Попов	149	152	155	159
Кеш. адапт. Попов	149	152	155	159
Адапт. Маліцький Tam	171	175	178	182
Кеш. адапт. Маліцький Tam	171	175	178	182

Таблиця 4.8: Число ітерацій

Ситуація доволі схожа на попередню, за виключення того що алгоритм Попва тепер не так суттєво програє неадаптивній версії.

4.5 Друга задача, неадаптивні алгоритми

Для кожного алгоритму і кожного розміру задачі було проведено 5 запусків (із різними матрицями), у таблиці і на графіку наведені середні значення та середньоквадратичні відхилення.

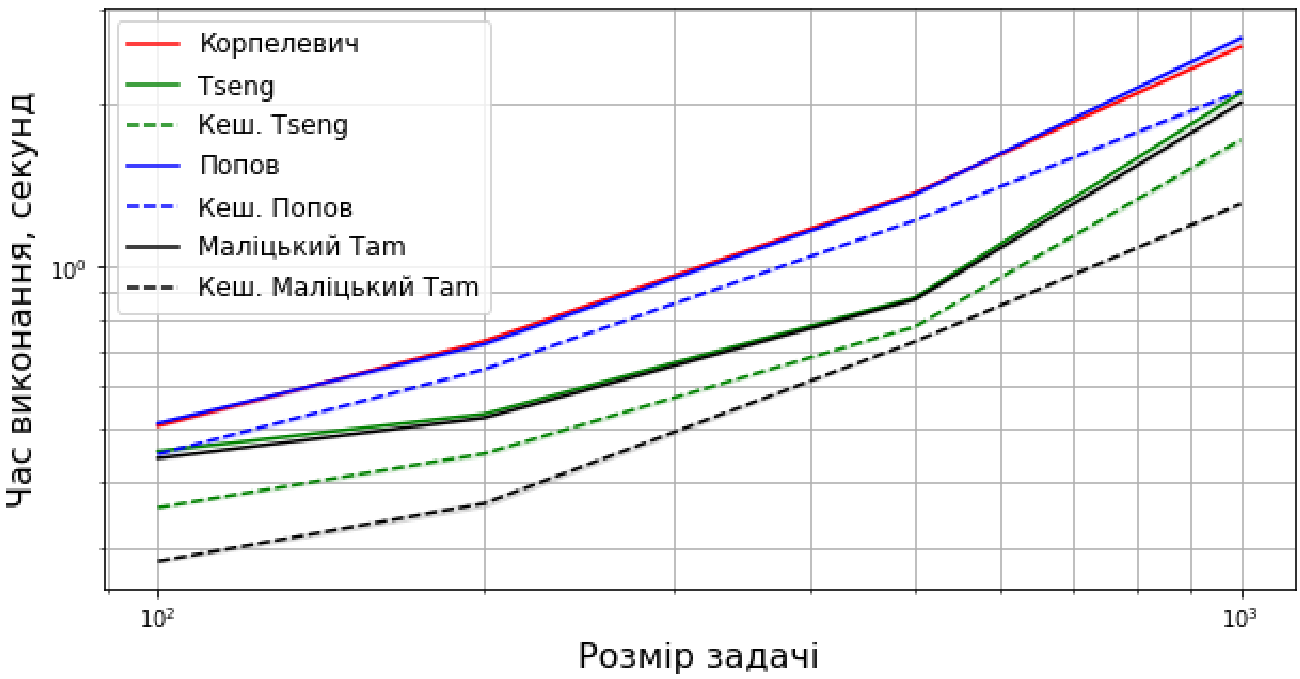


Рис. 4.5: Результати неадаптивних алгоритмів на другій задачі

Та сама інформація у табличці, для зручності:

Розмір задачі	100	200	500	1000
Корпелевич	0.51 ± 0.01	0.73 ± 0.02	1.37 ± 0.02	2.56 ± 0.03
Tseng	0.46 ± 0.02	0.53 ± 0.01	0.88 ± 0.02	2.10 ± 0.13
Кеш. Tseng	0.36 ± 0.01	0.45 ± 0.02	0.78 ± 0.01	1.72 ± 0.09
Попов	0.51 ± 0.02	0.72 ± 0.02	1.36 ± 0.02	2.66 ± 0.22
Кеш. Попов	0.45 ± 0.02	0.65 ± 0.01	1.22 ± 0.01	2.12 ± 0.10
Маліцький Тат	0.44 ± 0.01	0.53 ± 0.01	0.87 ± 0.02	2.02 ± 0.07
Кеш. Маліцький Тат	0.28 ± 0.01	0.36 ± 0.02	0.73 ± 0.01	1.31 ± 0.03

Таблиця 4.9: Час виконання, секунд

У цій задачі основна складність все ще у обчисленні оператора A, хоча обчислення проекції вже більш складне, тому алгоритм Tseng'а має певну перевагу над алгоритмом Попова, який у свою чергу випереджає алгоритм Корпелевич. Щодо кількості ітерацій то усі три алгоритми демонструють практично ідентичні результати.

Розмір задачі	100	200	500	1000
Корпелевич	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0
Tseng	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0
Кеш. Tseng	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0
Попов	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0
Кеш. Попов	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0
Маліцький Tam	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0
Кеш. Маліцький Tam	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0

Таблиця 4.10: Число ітерацій

Знову ж таки, кешування дає перевагу на великих задачах, хоча вона вже не у 1.5–2 рази.

Зауваження. Можна додати якийсь із матрчиних розкладів M для пришвидшення множення Mx.

4.6 Друга задача, адаптивні алгоритми

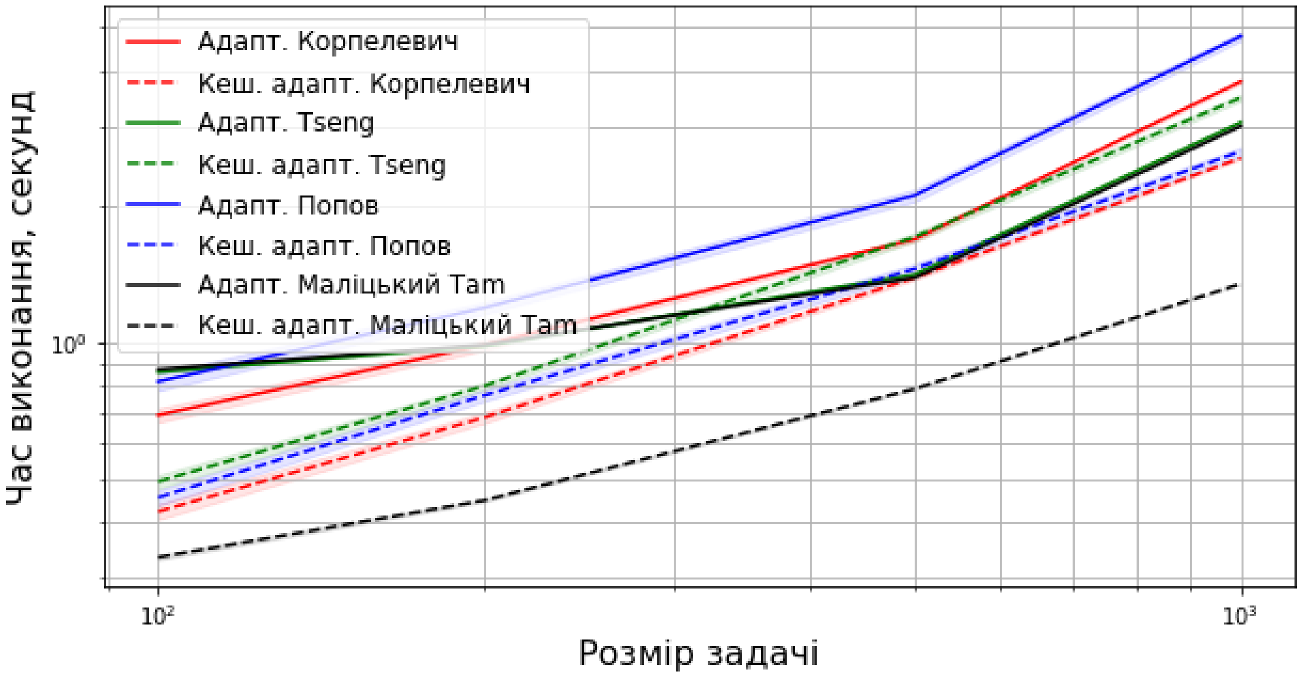


Рис. 4.6: Результати адаптивних алгоритмів на другій задачі

Та сама інформація у табличці:

Розмір задачі	100	200	500	1000
Адапт. Корпелевич	0.69 ± 0.12	0.99 ± 0.13	1.70 ± 0.10	3.79 ± 0.15
Кеш. адапт. Корпелевич	0.42 ± 0.08	0.68 ± 0.10	1.39 ± 0.07	2.56 ± 0.07
Адапт. Tseng	0.86 ± 0.02	0.98 ± 0.01	1.42 ± 0.02	3.08 ± 0.11
Кеш. адапт. Tseng	0.49 ± 0.09	0.80 ± 0.07	1.72 ± 0.19	3.49 ± 0.29
Адапт. Попов	0.82 ± 0.19	1.19 ± 0.18	2.12 ± 0.33	4.78 ± 0.51
Кеш. адапт. Попов	0.45 ± 0.09	0.76 ± 0.11	1.46 ± 0.19	2.65 ± 0.28
Адапт. Маліцький Tam	0.87 ± 0.02	0.99 ± 0.03	1.39 ± 0.03	3.02 ± 0.02
Кеш. адапт. Маліцький Tam	0.33 ± 0.02	0.45 ± 0.01	0.79 ± 0.01	1.35 ± 0.00

Таблиця 4.11: Час виконання, секунд

Адаптивні версії суттєво випереджають неадаптивні, причому за числом ітерацій також:

Розмір задачі	100	200	500	1000
Адапт. Корпелевич	763 ± 144	896 ± 125	957 ± 52	987 ± 25
Кеш. адапт. Корпелевич	763 ± 144	896 ± 125	957 ± 52	987 ± 25
Адапт. Tseng	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0
Кеш. адапт. Tseng	1193 ± 200	1568 ± 153	1983 ± 239	2109 ± 170
Адапт. Попов	909 ± 206	1086 ± 166	1149 ± 146	1239 ± 118
Кеш. адапт. Попов	909 ± 206	1086 ± 166	1149 ± 146	1239 ± 118
Адапт. Маліцький Tam	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0
Кеш. адапт. Маліцький Tam	1000 ± 0	1000 ± 0	1000 ± 0	1000 ± 0

Таблиця 4.12: Число ітерацій

4.7 Четверта задача, адаптивні алгоритми

А цій задачі константа Ліпшиця мені невідома, тому тут наводяться результати лише адаптивних алгоритмів.

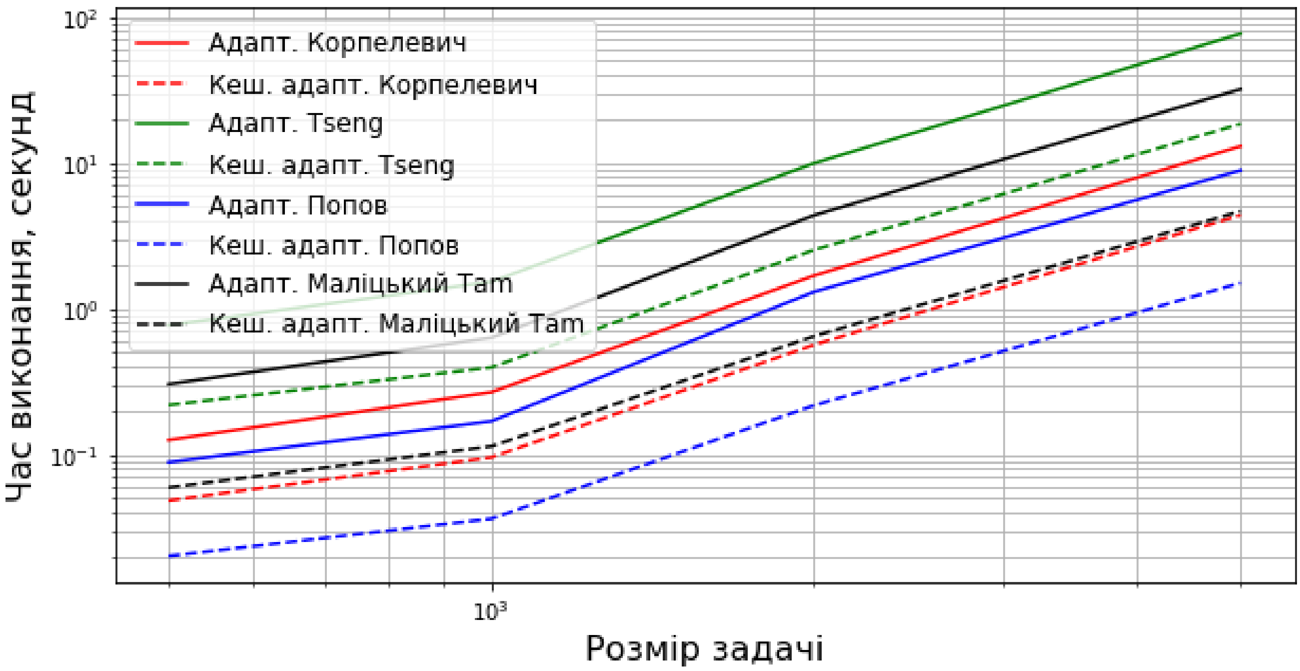


Рис. 4.7: Результати адаптивних алгоритмів на четвертій задачі

Та сама інформація у табличці:

Розмір задачі	500	1000	2000	5000
Адапт. Корпелевич	0.13	0.27	1.68	12.99
Кеш. адапт. Корпелевич	0.05	0.10	0.56	4.38
Адапт. Tseng	0.77	1.53	9.93	76.85
Кеш. адапт. Tseng	0.22	0.39	2.52	18.50
Адапт. Попов	0.09	0.17	1.30	8.86
Кеш. адапт. Попов	0.02	0.04	0.22	1.50
Адапт. Маліцький Tam	0.30	0.63	4.36	31.98
Кеш. адапт. Маліцький Tam	0.06	0.11	0.65	4.66

Таблиця 4.13: Час виконання, секунд

Розмір задачі	500	1000	2000	5000
Адапт. Корпелевич	111	113	116	119
Кеш. адапт. Корпелевич	111	113	116	119
Адапт. Tseng	558	572	587	605
Кеш. адапт. Tseng	463	477	492	510
Адапт. Попов	73	75	77	80
Кеш. адапт. Попов	73	75	77	80
Адапт. Маліцький Tam	232	238	243	251
Кеш. адапт. Маліцький Tam	232	238	243	251

Таблиця 4.14: Число ітерацій

4.8 Четверта задача із розрідженими матрицями, адаптивні алгоритми

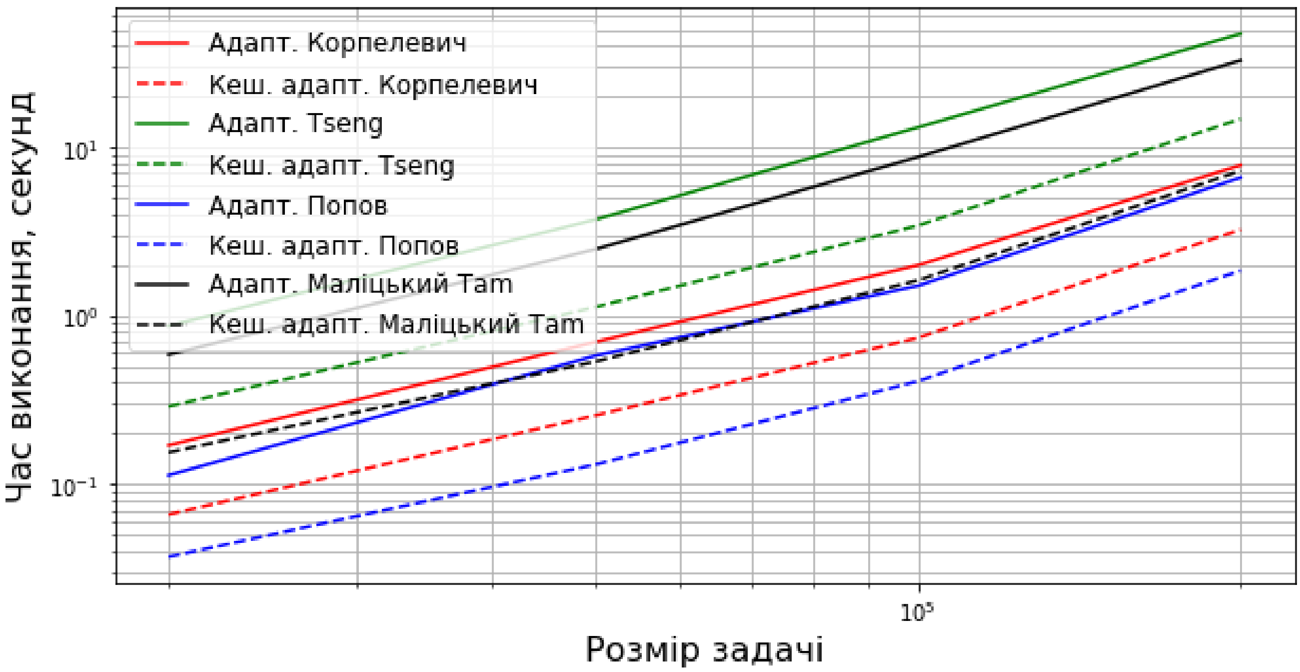


Рис. 4.8: Результати адаптивних алгоритмів на четвертій задачі із розрідженими матрицями

Та сама інформація у табличці:

Розмір задачі	20000	50000	100000	200000
Адапт. Корпелевич	0.17	0.70	1.99	7.82
Кеш. адапт. Корпелевич	0.07	0.26	0.74	3.24
Адапт. Tseng	0.87	3.72	13.12	47.17
Кеш. адапт. Tseng	0.29	1.12	3.43	14.72
Адапт. Попов	0.11	0.58	1.51	6.61
Кеш. адапт. Попов	0.04	0.13	0.41	1.85
Адапт. Маліцький Tam	0.59	2.48	8.74	32.78
Кеш. адапт. Маліцький Tam	0.15	0.53	1.62	7.25

Таблиця 4.15: Час виконання, секунд

Розмір задачі	20000	50000	100000	200000
Адапт. Корпелевич	74	76	77	79
Кеш. адапт. Корпелевич	74	76	77	79
Адапт. Tseng	388	399	408	416
Кеш. адапт. Tseng	332	343	352	360
Адапт. Попов	61	63	65	66
Кеш. адапт. Попов	61	63	65	66
Адапт. Маліцький Tam	262	269	275	280
Кеш. адапт. Маліцький Tam	262	269	275	280

Таблиця 4.16: Число ітерацій

5 Реалізація

Наведемо реалізацію усіх згаданих алгоритмів на мові програмування python.

Зауваження. Ми обрали дизайн згідно з яким власне алгоритм знає мінімальний контекст задачі. Це означає, що для використання алгоритму користувач має визначити дві функції, одна з яких відповідатиме за обчислення оператора A, а друга — за обчислення оператора P_C. Це надає користувачеві гнучкість у плані вибору способу обчислення операторів, яка буде помітна вже з перших тестових запусків.

Загальний вигляд (за модулем назви і деяких параметрів) запуску алгоритма наступний:

```python
solution, iteration_n, duration = korpelevich(
    x_initial=np.ones(size), lambda_=0.4,
    operator=lambda x: a.dot(x), projector=lambda x: x,
    tolerance=1e-3, max_iterations=1e4)
```

Як бачимо, визначення способу обчислення операторів A і P_C лягає на плечі користувача. У багатьох випадках це доволі просто, хоча у деяких користувачеві доведеться написати більше коду і знадобиться користуватися scipy.optimize або аналогічним модулем для обчислення проекції.

Ось, наприклад, клієнтський код для другої задачі:

```python
def ProjectionOntoProbabilitySymplex(x: np.array) -> np.array:
    dimensionality = x.shape[0]
    x /= dimensionality
    sorted_x = np.flip(np.sort(x))
    prefix_sum = np.cumsum(sorted_x)
    to_compare = sorted_x + (1 - prefix_sum) / np.arange(1, dimensionality + 1)
    k = 0
    for j in range(1, dimensionality): if to_compare[j] > 0: k = j
    return dimensionality * np.maximum(np.zeros(dimensionality),
                                       x + (to_compare[k] - sorted_x[k]))

solution, iteration_n, duration = korpelevich(...
    operator=lambda x: M.dot(x) + q,
    projector=ProjectionOntoProbabilitySymplex, ...)
```

5.1 Класичні алгоритми

Корпелевич

```
def korpelevich(x_initial: T,
                lambda_: float,
                operator: Callable[[T], T],
                projector: Callable[[T], T],
                tolerance: float = 1e-6,
                max_iterations: int = 1e3,
                **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None

    while True:
        # step 1
        y_current = projector(x_current - lambda_ * operator(x_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        x_next = projector(x_current - lambda_ * operator(y_current))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
```

Tseng

```python
def tseng(x_initial: T,
          lambda_: float,
          operator: Callable[[T], T],
          projector: Callable[[T], T],
          tolerance: float = 1e-6,
          max_iterations: int = 1e3,
          **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None

    while True:
        # step 1
        y_current = projector(x_current - lambda_ * operator(x_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        x_next = y_current - \
            lambda_ * (operator(y_current) - operator(x_current))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
```

Кешований Tseng

```
def cached_tseng(x_initial: T,
                 lambda_: float,
                 operator: Callable[[T], T],
                 projector: Callable[[T], T],
                 tolerance: float = 1e-6,
                 max_iterations: int = 1e3,
                 **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None

    while True:
        # step 1
        operator_x_current = operator(x_current)
        y_current = projector(x_current - lambda_ * operator_x_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        x_next = y_current - \
            lambda_ * (operator(y_current) - operator_x_current)

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
```

Попов

```
def popov(x_initial: T,
          y_initial: T,
          lambda_: float,
          operator: Callable[[T], T],
          projector: Callable[[T], T],
          tolerance: float = 1e-6,
          max_iterations: int = 1e3,
          **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_previous, y_current = y_initial, None

    while True:
        # step 1
        y_current = projector(x_current - lambda_ * operator(y_previous))

        # step 2
        x_next = projector(x_current - lambda_ * operator(y_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance and \
            norm(x_next - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
```

Кешований Попов

```python
def cached_popov(x_initial: T,
                 y_initial: T,
                 lambda_: float,
                 operator: Callable[[T], T],
                 projector: Callable[[T], T],
                 tolerance: float = 1e-6,
                 max_iterations: int = 1e3,
                 **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_previous, y_current = y_initial, None
    operator_y_previous, operator_y_current = operator(y_previous), None

    while True:
        # step 1
        y_current = projector(x_current - lambda_ * operator_y_previous)

        # step 2
        operator_y_current = operator(y_current)
        x_next = projector(x_current - lambda_ * operator_y_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance and \
            norm(x_next - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
        operator_y_previous, operator_y_current = operator_y_current, None
```

5.2 Адаптивні алгоритми

Адаптивний Корпелевич

```
def adaptive_korpelevich(x_initial: T,
                         tau: float,
                         lambda_initial: float,
                         operator: Callable[[T], T],
                         projector: Callable[[T], T],
                         tolerance: float = 1e-6,
                         max_iterations: int = 1e3,
                         **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None
    lambda_current, lambda_next = lambda_initial, None

    while True:
        # step 1
        y_current = projector(x_current - lambda_current * operator(x_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        x_next = projector(x_current - lambda_current * operator(y_current))

        # step 3
        if (operator(x_current) - operator(y_current)).dot(x_next - y_current) <= 0:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau / 2 *
                (np.linalg.norm(x_current - y_current) ** 2 +
                 np.linalg.norm(x_next - y_current) ** 2) /
                   (operator(x_current) - operator(y_current)).dot(x_next - y_current))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
        lambda_current, lambda_next = lambda_next, None
```

Кешований адаптивний Корпелевич

```
                                tau: float,
                                lambda_initial: float,
                                operator: Callable[[T], T],
                                projector: Callable[[T], T],
                                tolerance: float = 1e-6,
                                max_iterations: int = 1e3,
                                **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None
    lambda_current, lambda_next = lambda_initial, None
    operator_x_current, operator_y_current = None, None

    while True:
        # step 1
        operator_x_current = operator(x_current)
        y_current = projector(x_current - lambda_current * operator_x_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        operator_y_current = operator(y_current)
        x_next = projector(x_current - lambda_current * operator_y_current)

        # step 3
        product = (operator_x_current - operator_y_current).dot(x_next - y_current)
        if product <= 0:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau / 2 *
                (np.linalg.norm(x_current - y_current) ** 2 +
                 np.linalg.norm(x_next - y_current) ** 2) / product)

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
        lambda_current, lambda_next = lambda_next, None
        operator_x_current, operator_y_current = None, None
```

Адаптивний Tseng

```python
def adaptive_tseng(x_initial: T,
                   tau: float,
                   lambda_initial: float,
                   operator: Callable[[T], T],
                   projector: Callable[[T], T],
                   tolerance: float = 1e-6,
                   max_iterations: int = 1e3,
                   **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None
    lambda_current, lambda_next = lambda_initial, None

    while True:
        # step 1
        y_current = projector(x_current - lambda_current * operator(x_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        x_next = y_current - \
            lambda_current * (operator(y_current) - operator(x_current))

        # step 3
        if norm(operator(x_current) - operator(y_current)) < tolerance:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau *
                np.linalg.norm(x_current - y_current) /
                np.linalg.norm(operator(x_current) - operator(y_current)))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
        lambda_current, lambda_next = lambda_next, None
```

Кешований адаптивний Tseng

```python
def cached_adaptive_tseng(x_initial: T,
                          tau: float,
                          lambda_initial: float,
                          operator: Callable[[T], T],
                          projector: Callable[[T], T],
                          tolerance: float = 1e-5,
                          max_iterations: int = 1e4,
                          **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None
    lambda_current, lambda_next = lambda_initial, None
    operator_x_current, operator_y_current = None, None

    while True:
        # step 1
        operator_x_current = operator(x_current)
        y_current = projector(x_current - lambda_current * operator_x_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        operator_y_current = operator(y_current)
        x_next = y_current - \
            lambda_current * (operator_y_current - operator_x_current)

        # step 3
        if norm(operator_x_current - operator_y_current) < tolerance:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau *
                np.linalg.norm(x_current - y_current) /
                np.linalg.norm(operator_x_current - operator_y_current))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
        lambda_current, lambda_next = lambda_next, None
        operator_x_current, operator_y_current = None, None
```

Адаптивний Попов

```python
def adaptive_popov(x_initial: T,
                   y_initial: T,
                   tau: float,
                   lambda_initial: float,
                   operator: Callable[[T], T],
                   projector: Callable[[T], T],
                   tolerance: float = 1e-5,
                   max_iterations: int = 1e4,
                   **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_previous, y_current = y_initial, None
    lambda_current, lambda_next = lambda_initial, None

    while True:
        # step 1
        y_current = projector(x_current - lambda_current * operator(y_previous))

        # step 2
        x_next = projector(x_current - lambda_current * operator(y_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance and \
            norm(x_next - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 3
        if (operator(y_previous) - operator(y_current)).dot(x_next - y_current) <= 0:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau / 2 *
                (norm(y_previous - y_current) ** 2 +
                 norm(x_next - y_current) ** 2) /
                    (operator(y_previous) - operator(y_current)).dot(x_next - y_current))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
        lambda_current, lambda_next = lambda_next, None
```

Кешований адаптивний Попов

```
def cached_adaptive_popov(x_initial: T,
                          y_initial: T,
                          tau: float,
                          lambda_initial: float,
                          operator: Callable[[T], T],
                          projector: Callable[[T], T],
                          tolerance: float = 1e-5,
                          max_iterations: int = 1e4,
                          **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_previous, y_current = y_initial, None
    lambda_current, lambda_next = lambda_initial, None
    operator_y_previous, operator_y_current = operator(y_previous), None

    while True:
        # step 1
        y_current = projector(x_current - lambda_current * operator_y_previous)

        # step 2
        operator_y_current = operator(y_current)
        x_next = projector(x_current - lambda_current * operator_y_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance and \
            norm(x_next - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 3
        product = (operator_y_previous - operator_y_current).dot(x_next - y_current)
        if product <= 0:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau / 2 *
                (norm(y_previous - y_current) ** 2 +
                 norm(x_next - y_current) ** 2) / product)

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
        lambda_current, lambda_next = lambda_next, None
        operator_y_previous, operator_y_current = operator_y_current, None
```

5.3 Алгоритм Маліцького—Там'а

Маліцький—Там

```
def malitskyi_tam(x0_initial: T,
                  x1_initial: T,
                  lambda_: float,
                  operator: Callable[[T], T],
                  projector: Callable[[T], T],
                  tolerance: float = 1e-6,
                  max_iterations: int = 1e3,
                  **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_previous, x_current, x_next = x0_initial, x1_initial, None

    while True:
        # step
        x_next = projector(x_current - lambda_ * operator(x_current) -
            lambda_ * (operator(x_current) - operator(x_previous)))

        # stopping criterion
        if norm(x_current - x_previous) < tolerance and \
            norm(x_next - x_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # next iteration
        iteration_number += 1
        x_previous, x_current, x_next = x_current, x_next, None
```

Кешований Маліцький—Там

```
def cached_malitskyi_tam(x0_initial: T,
                         x1_initial: T,
                         lambda_: float,
                         operator: Callable[[T], T],
                         projector: Callable[[T], T],
                         tolerance: float = 1e-6,
                         max_iterations: int = 1e3,
                         **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_previous, x_current, x_next = x0_initial, x1_initial, None
    operator_x_previous, operator_x_current = \
        operator(x_previous), operator(x_current)

    while True:
        # step
        x_next = projector(x_current - lambda_ * operator_x_current -
            lambda_ * (operator_x_current - operator_x_previous))

        # stopping criterion
        if norm(x_current - x_previous) < tolerance and \
            norm(x_next - x_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # next iteration
        iteration_number += 1
        operator_x_previous, operator_x_current = \
            operator_x_current, operator(x_next)
        x_previous, x_current, x_next = x_current, x_next, None
```

Адаптивний Маліцький—Tam

```
def adaptive_malitskyi_tam(x0_initial: T,
                           x1_initial: T,
                           tau: float,
                           lambda0_initial: float,
                           lambda1_initial: float,
                           operator: Callable[[T], T],
                           projector: Callable[[T], T],
                           tolerance: float = 1e-6,
                           max_iterations: int = 1e3,
                           **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_n = 1
    x_previous, x_current, x_next = x0_initial, x1_initial, None
    lambda_previous, lambda_current, lambda_next = \
        lambda0_initial, lambda1_initial, None

    while True:
        # step 1
        x_next = projector(x_current - lambda_current * operator(x_current) -
            lambda_previous * (operator(x_current) - operator(x_previous)))

        # stopping criterion
        if norm(x_current - x_previous) < tolerance and \
            norm(x_next - x_current) < tolerance or \
            iteration_n == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_n, duration

        # step 2
        if norm(operator(x_next) - operator(x_current)) < tolerance:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau *
                norm(x_next - x_current) /
                norm(operator(x_next) - operator(x_current)))

        # next iteration
        iteration_n += 1
        x_previous, x_current, x_next = x_current, x_next, None
        lambda_previous, lambda_current, lambda_next = \
            lambda_current, lambda_next, None
```

Кешований адаптивний Маліцький—Там

```python
def cached_adaptive_malitskyi_tam(x0_initial: T,
                                  x1_initial: T,
                                  tau: float,
                                  lambda0_initial: float,
                                  lambda1_initial: float,
                                  operator: Callable[[T], T],
                                  projector: Callable[[T], T],
                                  tolerance: float = 1e-6,
                                  max_iterations: int = 1e3,
                                  **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_n = 1
    x_previous, x_current, x_next = x0_initial, x1_initial, None
    lambda_previous, lambda_current, lambda_next = \
        lambda0_initial, lambda1_initial, None
    operator_x_previous, operator_x_current, operator_x_next = \
        operator(x_previous), operator(x_current), None

    while True:
        # step 1
        x_next = projector(x_current - lambda_current * operator_x_current -
            lambda_previous * (operator_x_current - operator_x_previous))

        # stopping criterion
        if norm(x_current - x_previous) < tolerance and \
            norm(x_next - x_current) < tolerance or \
            iteration_n == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_n, duration

        # step 2
        operator_x_next = operator(x_next)
        if norm(operator_x_next - operator_x_current) < tolerance:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau *
                norm(x_next - x_current) /
                norm(operator_x_next - operator_x_current))

        # next iteration
        iteration_n += 1
        x_previous, x_current, x_next = x_current, x_next, None
        lambda_previous, lambda_current, lambda_next = \
            lambda_current, lambda_next, None
        operator_x_previous, operator_x_current, operator_x_next = \
            operator_x_current, operator_x_next, None
```

6 TODO

1. problem #3
2. runner
3. реферат
4. джерела
5. презентація