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

Кваліфікаційна робота

на здобуття ступеня магістра

за спеціальністю 113 Прикладна математика
на тему:

Деякі оптимізаційні задачі міського планування

Виконав студент 2-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 — 2022

РЕФЕРАТ

Обсяг роботи 26 сторінок, 2 ілюстрації, 2 таблиці, 24 джерела посилань, 1 додаток.

МІСЬКЕ ПЛАНУВАННЯ, ТРАНСПОРТНЕ ПЛАНУВАННЯ, ПОТОКОВІ МЕРЕЖІ, ВИПАДКОВІ ГРАФИ.

Об’єктом роботи є процес постановки потокових задач стабільності та їхнього розв’язування за допомогою власноруч розробленого програмного засобу. Предметом роботи є власноруч розроблений алгоритм та відповідний програмний засіб розв’язування потокових задач стабільності.

Метою роботи є постановка потокових задач стабільності та розробка програмного засобу їхнього розв’язування на базі нових алгоритмів, тестування програмного засобу на модельних задачах, оцінка і аналіз результатів.

Методи розроблення: розробка програмного продукту, комп’ютерне моделювання, чисельний експеримент. Інструменти розроблення: текстовий редактор Sublime Text 3, текстовий редактор Vim, компілятор Clang, Jupyter Notebook, мова програмування Python, мова програмування C++.

Результати роботи: поставлені потокові задачі стабільності, розроблено алгоритми їхнього розв’язування та відповідний програмний засіб, виконано тестування програмного засобу на модельних задачах, проведена оцінка і аналіз результатів.

ЗМІСТ

1 Вступ

1.1 Що таке міське планування?

Містобудування чи міське планування — комплексна багатогранна діяльність суспільства, що спрямована на створення матеріально-просторового середовища життєдіяльності людини в поселеннях та районах розселення.

Традиційно, містобудування відбувалося за підходом згори-донизу, у якому генеральні плани визначали фізичне розташування тих чи інших будівель. Основною метою було забезпечення громадського добробуту, що включало ефективність використання ресурсів та середовища, соціальної та економічної діяльності суспільства.

З часом, міське планування адаптувалося і почалося розглядатися не лише як інструмент забезпечення сьогоденної ефективності, але і як засіб забезпечення *сталого розвитку*. Це особливо актуально з кінця 20-го сторіччя, адже негативний вплив людської діяльності на середовище стає дедалі більш явним. [23]

1.2 Галузі міського планування

До основних галузей міського планування належать транспорт, естетика, безпека, планування передмість, взаємодія з середовищем, зонування, водопостачання тощо. Практично усі аспекти соціальної діяльності людини можуть підлягати плануванню.

Однією з галузей є організація дорожнього руху. *Об'єктами галузі* є процес визначення політик, цілей, джерел фінансування, та, нарешті, власне просторового планування дорожніх мереж. *Метою галузі* є забезпечення майбутніх потреб у переміщенні людей, ресурсів та товарів. [22]

Без сумніву, транспортні мережі також мають забезпечувати можливість сталого розвитку. Втім, деякі люди розглядають це питання виключно з точки зору зменшення викидів вуглецевого газу за рахунок поширення електротранспорту. [21]

1.3 Стійкість транспортної мережі

Наступний історичний експонат яскраво демонструє тогочасну відсутність людських уявлень про транспортну мережу, котра змогла б витримати постійний розвиток (надалі *стійку* транспортну мережу):

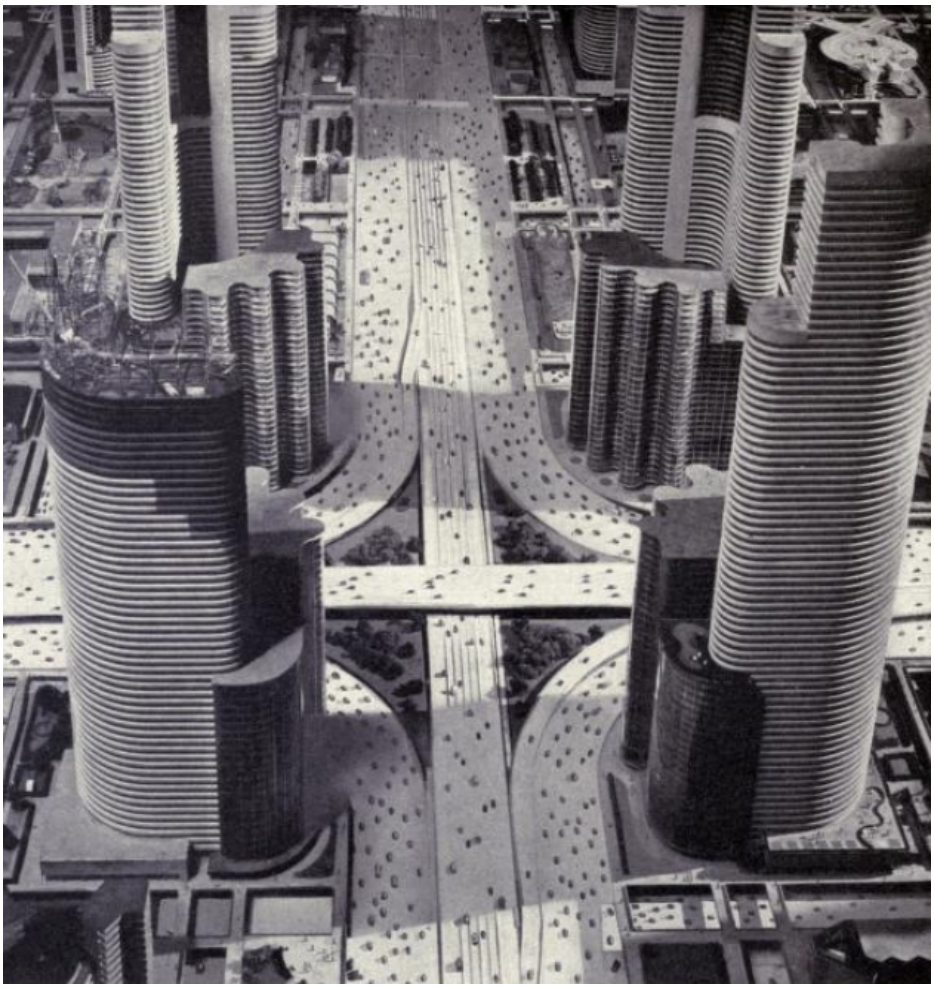

Рис. 1.1: “Футурама” — спонсорований корпорацією “Дженерал моторс” експонат з всесвітньої виставки 1939-го року у Нью-Йорку, що зображав тогочасне бачення міста майбутнього.

Наразі стало зрозуміло, що подібна інтенсивність дорожнього руху та власне дорожньої забудови неприпустима і може хіба що нашкодити транспортній мережі. [12]. На жаль, методи розробки транспортних мереж з перспективами розвитку залишаються радше *еврестичними*.

Критеріями оцінки стійкості транспортної мережі виступають конкретні транспортні засоби, джерела енергії, та інфраструктура, що забезпечує транспортування (дороги, залізниці, авіалінії, канали тощо).

2 Задачі

2.1 Мотивація

Багато класичних задач транспортного планування можуть бути сформульовані у термінах *потоків* та пов'язаних із ними задач: максимальний потік, максимальний потік мінімальної вартості, задача про циркуляцію тощо. [17, 18, 6]

Зазначимо також, що у термінах потоків можна сформулювати такі класичні задачі як парування у дводольному графі та задачу про призначення. [16, 8]

Давно розроблені та добре відомі алгоритми, що розв'язують ці задачі — алгоритми Форда—Фалкерсона, Едмондса—Карпа, Дініца, алгоритм проштовхування передпотоку та багато його варіантів. [7, 4, 24, 9].

Втім, у всіх згаданих алгоритмів є спільна риса, що може стати критичною вадою у задачі пошуку стійкої транспортної мережі. А саме, усі класичні алгоритми спрямовані на знаходження оптимального розв'язку конкретної задачі, нехтуючи *адаптивністю* шуканого розв'язку.

Наприклад, складно передбачити зміну максимального потоку при додаванні чи прибиранні кількох вершин та ребер мережі. Фахівці з машинного навчання сказали б, що класичні алгоритми схильні до *перенавчання*. [19]

Для формалізації процесу додавання і прибирання вершин та ребер нам знадобляться деякі поняття з теорії випадкових графів. Випадкові графи є нічим іншим, як вибіркою з ймовірнісного розподілу на множині графів.

У літературі визначено багато різних моделей випадкових графів, що, по суті, відповідають різним розподілам, такі як модель Ердеша—Реньї, ієрархічна модель тощо. [5, 15]

Для оцінки адаптивності того чи іншого алгоритму пропонуємо наступну схему: спершу запускаємо алгоритм на заданій мережі, далі генеруємо певну кількість випадкових варіацій початкової мережі і просимо алгоритм доповнити раніше знайдений розв'язок. Чим кращий середній результат алгоритма на випадковій вибірці варіацій, тим адаптивнішим вважатимемо алгоритм.

2.2 Мережі та потоки

Введемо деякі поняття з галузі потокових мереж. [14]

Мережею будемо називати граф $G = (V, E)$, де V – множина вершин, а E – множина ребер, що є підмножиною $V \times V$, разом із невід'ємною функцією $c : V \times V \to \mathbb{R}_\infty$, котра називається функцією *ємності*. Без обмеження загальності, можна вважати, що якщо $(u, v) \in E$ то і $(v, u) \in E$, довизначивши при цьому $c(v, u) = 0$.

Якщо також окремо виділить дві вершини у G – *джерело* s та *сток* t – то отримаємо *потокову мережу* (G, c, s, t).

Є кілька способів ввести у потоковій мережі потік. Найпростішим є так званий *псевдо-потік*. Псведо-потік це функція $f : V \times V \to \mathbb{R}$, що задовольняє наступні обмеження для усіх пар вершин u та v:

- кососиметричність: $f(u, v) = -f(v, u)$;
- обмеження ємності: $f(u, v) \leq c(u, v)$.

Маючи псевдо-потік f часто буває корисним розглянути сумарну величину потоку, що входить у певну вершину u. Визначимо *функцію надлишку* $x_f : V \to \mathbb{R}$ як $x_f(u) = \sum_{v \in V} f(v, u)$. Вершину u назвемо *активною*, якщо $x_f(u) > 0$, *дефіцитною*, якщо $x_f(u) < 0$, та *зберігаючою*, якщо $x_f(u) = 0$.

Нововведені означення дозволяють посилити визначення псевдо-потоку.

Передпотоком будемо називати псевдо-потік, який задовольняє наступній додатковій умові, для усіх вершин $v \in V$ окрім джерела s:

- недефіцитність: $x_f(v) \geq 0$.

Нарешті, *потоком* називають псевдо-потік, що для усіх вершин $v \in V$ окрім джерела s та стоку t задовольняє наступній умові:

- збереження потоку: $x_f(v) = 0$.

Величиною потоку f (позначається $|f|$) називається сумарний потік, що входить у сток, тобто $|f| = x_f(t)$.

Для подальшого заглиблення у теорію потоків рекомендуємо ознайомитися з такими ресурсами як [2] та [11].

2.3 Випадкові графи

Введемо деякі поняття з галузі випадкових графів. [20]

Випадковий граф можна отримати у результаті наступного процесу: починаємо з n ізольованих вершин і поступово додаємо випадкові ребра між ними. Метою досліджень цієї галузі є визначення моменту, на якому певна властивість графу стає ймовірною.

Різні моделі випадкових графів призводять до різних ймовірнісних розподілів. Найчастіше розглядається модель $G(n, p)$, у якій кожне ребро присутнє з ймовірністю $0 < p < 1$. Ймовірність отримання заданого графу з m ребрами становить $p^m(1-p)^{\binom{n}{2}-m}$.

Тісно пов'язана модель Ердеша—Реньї $G(n, m)$ приписує рівні ймовірності всім графам з M ребрами, де $0 \leq m \leq \binom{n}{2}$. У $G(n, m)$ рівно $\binom{\binom{n}{2}}{m}$ елементів, і ймовірність кожного з них становить $1/\binom{\binom{n}{2}}{m}$. Цю модель можна розглядати як знімок у момент часу m описаного вище стохастичного графового процесу.

Іншою моделлю, що узагальнює першу, є модель скалярного добутку. У ній кожній вершині приписаний дійсний вектор. Ймовірність ребра (u, v) між вершинами u та b є функцією скалярного добутку $\vec{u} \cdot \vec{v}$ відповідних векторів.

Також можна розглянути модель у якій задається матриця $p_{i,j}$, які позначають ймовірності входження ребра $e_{i,j}$ до випадкового графу. Цю модель можна розширити на орієнтовані та неорієнтовані, зважені та незважені, статичні та динамічні графові структури.

Важливо розуміти, що для $m \approx p\binom{n}{2}$ дві основні моделі практично не відрізняються.

Коли модель випадкового графу зафіксовано, кожна функція від графу стає випадковою величиною. Метою дослідження моделі є визначення, або принаймні оцінка ймовірності певної випадкової події.

Для подальшого заглиблення у теорію випадкових графів радимо ознайомитися з таким ресурсом як [3].

2.4 Постановка задачі

Для простоти розглянемо задачу про максимальний потік у мережі $G \sim G(n, p_1)$ із $c(u, v) \sim U_{[0,1]}$ ($c(u, v) = 0$ якщо $(u, v) \notin E$). Для конкретної задачі цільовою функцією є величина потоку з джерела $s = 1$ до стоку $t = n$.

Далі розглянемо вибірку з k випадкових варіацій $G_1, \ldots, G_k$ мережі G. У G_i кожне ребро змінює свій стан (якщо було то зникає, якщо не було то з'являється) з ймовірністю $p_2 \ll p_1$. Окрім того $c_i \sim c \cdot U_{[1-r,1+r]}$, де $r \ll 1$ (далі c_i нормується на $[0, 1]$ жорстким порогуванням).

Зазначимо, що в аналізі даних та машинному навчанні подібний підхід доповнення даних давно відомий та добре себе зарекомендував якраз як механіз боротьби проти перенавчання. [13]

З практичної точки зору, параметр p_2 моделює зміни транспортної мережі у середній перспективі, як-то прокладання нових шляхів чи перекриття старих (наприклад, у зв'язку з дорожньо-ремонтними роботами). Параметр r моделює миттєві зміни транспортної мережі такі як автомобільні аварії, утворення заторів тощо.

Цільовою функцією задачі стабільності будемо вважати величину доповнюючого потоку, усереднену за вибіркою. *Доповнюючим потоком* будемо називати потік f' із $f'(u, v) \geq f(u, v)$.

2.5 З точки зору лінійного програмування

Зауважимо, що жорстка задача (у якій замість усереднення за варіаціями береться мінімум) не становить наукової новизни. Справді, задачу пошуку максимального потоку можна подати у вигляді загальної задачі лінійного програмування.

Якщо далі вимагати, аби розв'язок задовольняв усім можливим незначним варіаціям правих частих нерівностей, то допустимою множиною буде перетин "найстрогіших" варіантів усіх напівплощин. Тобто задача такого ж вигляду як початкова, просто з більш строгими обмеженнями.

Якщо також дозволити незначні варіації коефіцієнтів лівих частин, то допустимою областю буде певна опукла множина, і маємо загальну задачу опуклої оптимізації. На жаль, ці спостереження не дають поліноміального алгоритму.

3 Алгоритми

3.1 Незастосовність класичних методів

Лема 1. У задачі стабільності для максимального потоку f з ймовірністю $\geq 1 - (1/2)^{se(f)}$ не існує доповнюючого потоку, де $se(f)$ — число насичених ребер у f.

Доведення. Нагадаємо, що ребро (u, v) називається *насиченим* у потоці f, якщо $c(u, v) = f(u, v)$.

Далі, якщо $(u, v) \in f$, то з ймовірністю $1/2$ маємо $c'(u, v) < c(u, v)$, а далі з ланцюжка нерівностей $f(u, v) \leq f'(u, v) \leq c'(u, v) < c(u, v) = f(u, v)$ отримуємо суперечність для кожного насиченого ребра (u, v).

Оскільки ємності ребер незалежні, то за правилом добутку ємність жодного з насичених ребер не зменшиться з ймовірністю $(1/2)^{se(f)}$. Відповідно, ймовірність доповнюючої події становить $1 - (1/2)^{se(f)}$. □

Лема 2. Для максимального потоку f маємо $\lim_{n\to\text{inf}} \mathbb{E}_{G(n,\log n)}[se(f)] = \text{inf}$.

Доведення. Нагадаємо, що багато випадкових графів, що виникають у практичних ситуаціях, як-то соціальні та транспортні мережі, традиційно моделюють як $G(n, c \log n/n)$.

За теоремою про рівність максимального потоку та мінімального розрізу, $se(f) \geq mc$, де mc — кількість ребер у мінімальному $s - t$ розрізі, адже максимальний потік f зобов'язаний наситити усі ребра якогось розрізу.

Оцінимо знизу $\mathbb{E}_{G(n,\log n)}[mc]$. Нехай у s-долі n_s вершин, а у t-долі — $n_t = n - n_s$ вершин. Математичне сподівання числа ребер між ними, що ведуть з s-долі до t-долі дорівнює $n_s n_t \frac{\log n}{2n} \geq 1(n-1)\frac{\log n}{2n} \geq c \log n$. Отже $\mathbb{E}_{G(n,\log n)}[mc] \geq c \log n$, і твердження отримується граничним переходом при $n \to \text{inf}$. □

Теорема 1. У задачі стабільності ймовірність існування доповнюючого потоку для максимального потоку у $G(n, \log n/n)$ прямує до нуля при $n \to \text{inf}$.

Доведення. Безпосередньо випливає з двох попередніх лем. □

Таким чином ми показали, що для задач стабільності недоцільно використовувати класичні методи, котрі одразу знаходять максимальний потік, адже надалі його швидше за все не вийде доповнити.

З практичної точки зору це відповідає наступному спостереженню: якщо навіть у звичайний час дороги завантажені “під зав’язку”, то будь-які мінімальні коливання (ремонтні роботи, дпт, несправність світлофорів тощо) одразу призведуть до виведення мережі з ладу.

На практиці саме таке спостереження призвело до евристик вигляду “планувати дороги необхідно так, аби навантаження у плановій ситуації не перевищувало 70–80% від максимального”.

3.2 Неоптимальність відомих евристик

Зрозуміло, що 70–80% — чималий відсток, і здебільшого перестраховуються більше. Так, планове навантаження мостів та будівель зазвичай становить 10–20% від критичного. Давайте дослідимо, наскільки оптимальною є евристика у якій $f(u, v) \leq q \cdot c(u, v)$, де $0 < q < 1$. Для цього розглянемо, які ребра найчастіше насичує максимальний потік.

Застосуємо метод чисельного експерименту: згенеруємо багато випадкових потокових мереж за описаною вище процедурою, для кожної знайдемо максимальний потік, і обчислимо, скільки разів кожне ребро було насиченим. Програмний код цієї частини роботи наведений у додатку А.

При детальному розгляді результатів моделювання можна помітити, що найчастіше насичуються ребра (s, v) та (v, t), а решта ребер майже ніколи не насичуються. Такий разючий контраст показує, що не дуже оптимально обмежувати потік на усіх ребрах однією константою q.

n	p	$\lvert f\rvert$	$\#(s,t)$	$\#(u,v)$
10	.10	9.57	0.23	0.03
20	.10	30.12	0.83	0.55
30	.10	75.83	2.33	2.00
10	.30	75.79	2.22	0.72
20	.30	218.66	7.49	2.75
30	.30	357.14	12.71	4.13
10	.50	170.68	5.50	1.30
20	.50	402.22	14.27	3.16
30	.50	635.93	23.27	4.88
10	.70	264.16	8.84	1.34
20	.70	590.96	21.37	3.23
30	.70	925.26	34.38	5.19
10	.90	358.73	12.18	1.17
20	.90	788.04	28.76	2.98
30	.90	1223.86	45.70	4.87

Таблиця 3.1: Результати моделювання частоти насичення різних ребер випадкового графу. Через (s,t) позначено ребра вигляду (s,u) та (v,t), через (u,v) — усі інші. Наведені числа — середні значення відповідних величин. В усіх експериментах (рядках) було виконано щонайменше 20’000 ітерацій.

3.3 Пом’якшена стратегія

Натомість хотілося б строго обмежити лише ребра, ймовірність насичення яких висока, і сподіватися, що з рештою ребер проблем не виникне. Краще, звісно, не сподіватися а також їх обмежити, просто не так строго. Порівняємо дві стратегії:

- $f(u,v) \le (1-r)c(u,v)$;
- $f(s,v), f(v,t) \le (1-r)c(u,v)$ та $f(u,v) \le c(u,v) \le (1-qr)c(u,v)$.

Результати моделювання невтішні: пом’якшена стратегія стабільно програє. Якщо повернутися до попереднього чисельного експерименту, то можна з’ясувати причини: попри низьку ймовірність насичення кожного окремого ребра (u,v) із $u \ne s, v \ne t$ математичне сподівання числа таких ребер немаленьке. У пом’якшеній стратегії кожне насичене ребро переповнюється з невеликою ймовірністю $1/2 - q/2$, але навіть цього достатньо аби помітно позначитися на середній якості.

n	p	r	$\lvert f\rvert_{1-r}$	$\lvert f\rvert_{1-r,\ .9}$
10	.25	.05	52.76	47.55
20	.25	.05	158.30	114.55
30	.25	.05	253.47	159.30
10	.50	.05	159.46	124.21
20	.50	.05	375.69	197.93
30	.50	.05	597.28	209.92
10	.75	.05	273.76	184.94
20	.75	.05	604.51	214.27
30	.75	.05	951.53	175.46
10	.25	.10	50.93	48.47
20	.25	.10	145.45	124.41
30	.25	.10	232.66	184.40
10	.50	.10	157.26	138.97
20	.50	.10	347.41	255.78
30	.50	.10	567.69	338.19
10	.75	.10	261.96	216.03
20	.75	.10	569.74	351.94
30	.75	.10	882.32	398.64
10	.25	.20	45.22	42.70
20	.25	.20	123.23	102.36
30	.25	.20	221.93	157.83
10	.50	.20	131.69	111.27
20	.50	.20	326.95	209.64
30	.50	.20	510.19	249.13
10	.75	.20	221.52	173.56
20	.75	.20	508.96	259.61
30	.75	.20	775.62	264.33

Таблиця 3.2: Результати порівняння жорсткої та пом'якшеної стратегій. Наведені числа — середні значення відповідних величин. В усіх експериментах (рядках) було виконано щонайменше 20'000 ітерацій.

3.4 Декомпозиція потоку

Нагадаємо класичну теорему про декомпозицію потоку. [10]

Теорема 2. Будь-який потік f можна подати у вигляді сукупності $O(E)$ шляхів з джерела s до стоку t та циклів: $f = \sum_i c_i \cdot f(p_i) + \sum_j d_j \cdot f(k_j)$, де p_i — шлях з s до t, k_j — цикл, а c_i, d_j — деякі константи.

Доведення. Доведення проведемо спуском за числом ребер з ненульовим потоком.

Нехай з s виходить принаймні одне ребро з ненульовим потоком. Нехай це ребро (s, v_1). Якщо $v_1 = t$ то ми знайшли s-t шлях, інакше за законом збереження має знайтися ребро (v_1, v_2) з ненульовим потоком. Продовжимо цей процес допоки або $v_i = t$ (тоді знайшли шлях p) або $v_i = v_j$, $j < i$ (тоді знайшли цикл k).

Цей шлях (цикл) матиме ненульовий потік $f(p)$ ($f(k)$), що дорівнює мінімальному з потоків серед усіх своїх ребер. Зменшимо потік вздовж цього шляху (циклу) на цю величину, отримаємо новий потік. Зрозуміло, що при цьому потік вздовж принаймні одного ребра занулився, отже всього знадобиться не більше E таких операцій. □

Теорема про декомпозицію потоку дозволяє розглянути альтернативний підхід до генерації випадкового потоку. А саме, оберемо кілька випадкових шляхів і збільшим потік вздовж кожного з них на якусь випадкову величину. На жаль, для моделі Ердеша—Реньї такий підхід нічого не дає, адже усі проміжні вершини там рівноправні.

Натомість застосуємо цей підхід у більш практичній ситуації. Розглянемо множину цілих точок (x, y) таку, що $x^2 + y^2 \leq R^2$, з усіма ребрами довжини 1 (такі собі міські квартали). Ба більше, замість генерації випадкових шляхів між двома сталими точками будемо щоразу обирати дві випадкові точки і з'єднувати їх якимось з найкоротших шляхів (добре моделює поведінку окремих людей).

Проведемо ще один обчислювальний експеримент з метою визначення середньої інтенсивності руху у кожному кварталі. Як бачимо, у більш реалістичних умовах наше спостереження про різну завантаженість різних ділянок мережі залишається релевантним. Варто провести аналогічний експеримент з ієрархічною моделлю випадкового графу.

Прикро, що не вдалося одразу отримати разючих позитивних результатів, але негативний результат — теж результат. Навіть навпаки добре, що у предметній галузі "не все так просто". Це якраз означає, що є простір для подальших досліджень.

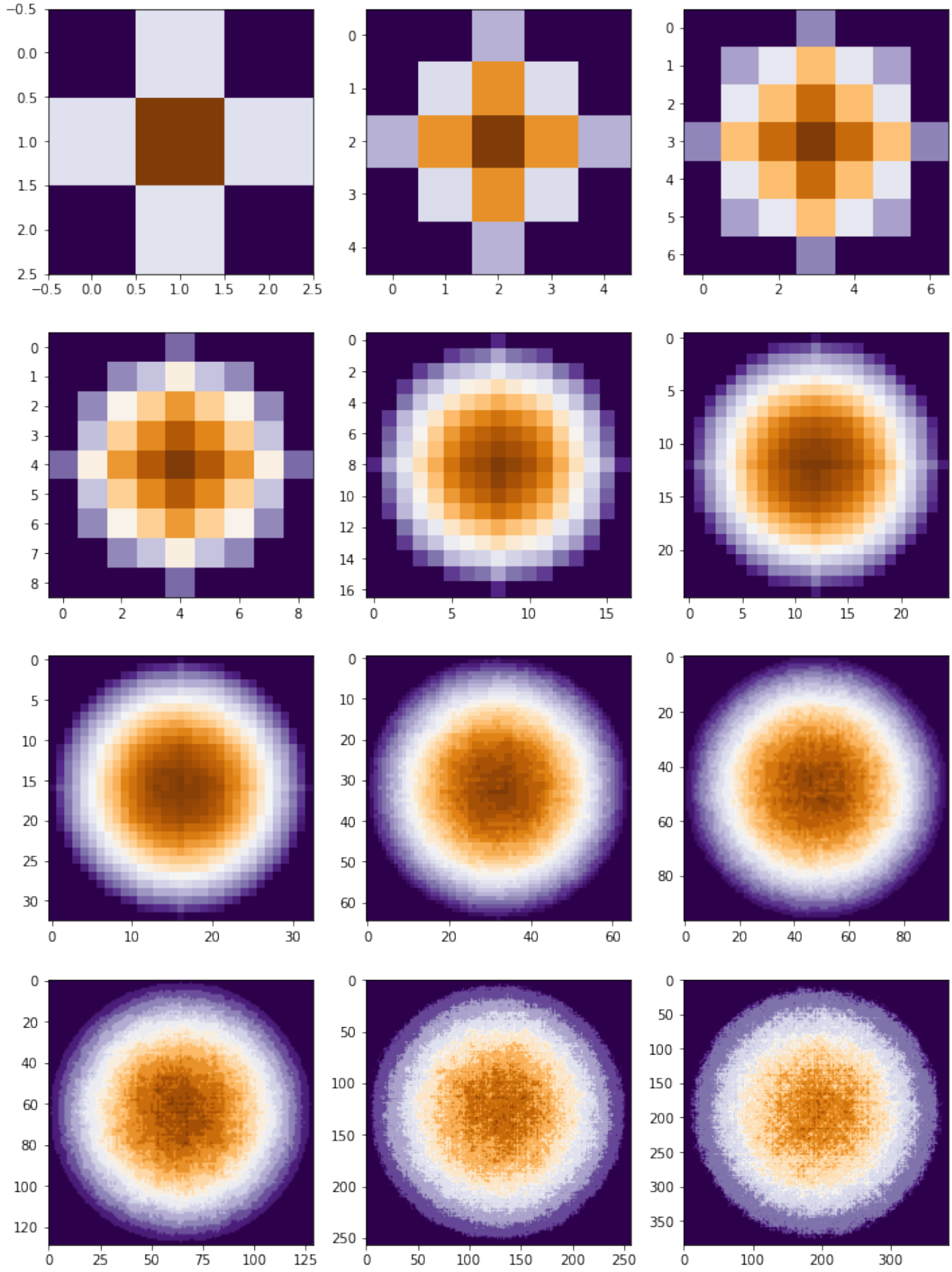

Рис. 3.1: Середня інтенсивність трафіку у кожному міському блоці, від фіолетового (низька) до помаранчевого (висока). В усіх експериментах (зображеннях) було виконано щонайменше 20’000 ітерацій.

4 ВИСНОВОК

В рамках дипломної роботи були виконані наступні завдання:

- розглянуто нові задачі стабільності для класичних потокових задач;
- розроблено методи та алгоритми розв’язання задач стабільності;
- виконано тестування алгоримтмів на широкому спектрі модельних задач;
- проведено детальний аналіз швидкодії і ефективності алгоритмів.

Визначено наступні напрямки майбутніх досліджень:

- замінити задачу пошуку максимального потоку задачею пошуку максимального потоку мінімальної вартості та задачею про циркуляцію;
- замінити Гільбертову модель випадкового графу моделями Ердеша—Реньї та ієрархічною моделлю;
- замінити спосіб генерації випадкових варіацій заданої потокової мережі;
- розглянути альтернативні підходи до розв’язання задач стабільності.

Окремо необхідно зазначити, що модель Барабаші—Альберт випадкового графу [1] становить особливий інтерес для планування регіонального та державного масштабу. Особливо перспективними для задач стабільності вважаю підходи, запозичені з машинного навчання, адже самі постановки задач походять з тієї галузі.

Безперечно, окрім справді випадкових графів необхідно також розглянути графи реальних транспортних мереж і порівняти результати моделювання. Окрім розв’язання власне поставлених задач це дозволить поглибити наші знання про практичну застосовність різних моделей випадкових графів.

Наостанок хочу сказати, що окрім потокових задач стабільності вважаю перспективними інші “нечіткі” задачі комбінаторної оптимізації. Для цього є практичний фундамент: зачасту ми або не знаємо точні обмеження задачі, або ми знайємо їх точно, але вони можуть змінюватися з часом (такий собі принцип невизначеності Гейзенберга для практичної комбінаторної оптимізації).

Бібліографія

[1] Albert, Reka; Barabasi, Albert-Laszlo. Statistical mechanics of complex networks / Albert, Reka; Barabasi, Albert-Laszlo // Reviews of Modern Physics. — 2002. — P. 47–97.

[2] Bela Bollobas. Graph Theory: An Introductory Course / Bela Bollobas. — Heidelberg: Springer-Verlag, 1979.

[3] Bela Bollobas. Random Graphs / Bela Bollobas. — Cambridge University Press, 2001.

[4] Edmonds, Jack; Karp, Richard M. Theoretical improvements in algorithmic efficiency for network flow problems / Edmonds, Jack; Karp, Richard M. // Journal of the ACM. — 1972. — P. 248–264.

[5] Erdos, P.; Renyi, A. On random graphs / Erdos, P.; Renyi, A. // Publicationes Mathematicae. — 1959. — P. 290–297.

[6] Eva Tardos. A strongly polynomial minimum cost circulation algorithm. — 1985.

[7] Ford, L. R.; Fulkerson, D. R. Maximal flow through a network / Ford, L. R.; Fulkerson, D. R. // Canadian Journal of Mathematics. — 1956. — P. 399–404.

[8] Fredman, Michael L.; Tarjan, Robert Endre. Fibonacci heaps and their uses in improved network optimization algorithms / Fredman, Michael L.; Tarjan, Robert Endre // ACM. — 1987. — P. 596–615.

[9] Goldberg, A. V.; Tarjan, R. E. A new approach to the maximum flow problem / Goldberg, A. V.; Tarjan, R. E. // Journal of the ACM. — 1988.

[10] Ravindra Ahuja, Thomas Magnanti, James Orlin. Network flows / Ravindra Ahuja, Thomas Magnanti, James Orlin. — Prentice-Hall, Inc., 1993.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms / Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. — MIT Press and McGraw-Hill, 2001.

[12] Wikipedia contributors. Carfree city — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 18-May-2022]. https://en.wikipedia.org/wiki/Carfree_city.

[13] Wikipedia contributors. Data augmentation — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 19-May-2022]. https://en.wikipedia.org/wiki/Data_augmentation.

[14] Wikipedia contributors. Flow network — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 19-May-2022]. https://en.wikipedia.org/wiki/Flow_network.

[15] Wikipedia contributors. Hierarchical network model — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 19-May-2022]. https://en.wikipedia.org/wiki/Hierarchical_network_model.

[16] Wikipedia contributors. Matching (graph theory): Bipartite matching — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 19-May-2022]. https://en.wikipedia.org/wiki/Matching_(graph_theory)#Bipartite_matching.

[17] Wikipedia contributors. Maximum flow problem — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 18-May-2022]. https://en.wikipedia.org/wiki/Maximum_flow_problem.

[18] Wikipedia contributors. Minimum-cost flow problem — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 19-May-2022]. https://en.wikipedia.org/wiki/Minimum-cost_flow_problem.

[19] Wikipedia contributors. Overfitting — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 19-May-2022]. https://en.wikipedia.org/wiki/Overfitting.

[20] Wikipedia contributors. Random graph — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 19-May-2022]. https://en.wikipedia.org/wiki/Random_graph.

[21] Wikipedia contributors. Sustainable transport: Environmental impact — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 18-May-2022]. https://en.wikipedia.org/wiki/Sustainable_transport#Environmental_impact.

[22] Wikipedia contributors. Transportation planning — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 18-May-2022]. https://en.wikipedia.org/wiki/Transportation_planning.

[23] Wikipedia contributors. Urban planning — Wikipedia, the free encyclopedia. — 2022. — [Online; accessed 18-May-2022]. https://en.wikipedia.org/wiki/Urban_planning.

[24] Yefim Dinitz. Algorithm for solution of a problem of maximum flow in a network with power estimation / Yefim Dinitz // Doklady Akademii Nauk SSSR. — 1970. — P. 1277–1280.

Додаток А Реалізація

Усі наведені фрагменти коду розміщені на GitHub за посиланням
https://github.com/nskybytskyi/random-maxflow
компілюються командою
`$ g++ -std=c++17 <source>.cpp -o <source>`
та запускаються командою
`$ ./<source> < <source>.in > <source>.out`

А.1 Алгоритм Дініца пошуку максимального потоку

```cpp
#ifndef MAXFLOW_HPP
#define MAXFLOW_HPP 1

#include <algorithm>
#include <cassert>
#include <limits>
#include <queue>
#include <vector>

template <class Cap> struct mf_graph {
  public:
    mf_graph() : _n(0) {}
    explicit mf_graph(int n) : _n(n), g(n) {}

    int add_edge(int from, int to, Cap cap) {
        int m = int(pos.size());
        pos.push_back({from, int(g[from].size())});
        int from_id = int(g[from].size());
        int to_id = int(g[to].size());
        if (from == to) to_id++;
        g[from].push_back(_edge{to, to_id, cap});
        g[to].push_back(_edge{from, from_id, 0});
        return m;
    }

    struct edge {
        int from, to;
        Cap cap, flow;
    };

    edge get_edge(int i) {
        int m = int(pos.size());
        auto _e = g[pos[i].first][pos[i].second];
        auto _re = g[_e.to][_e.rev];
        return edge{pos[i].first, _e.to, _e.cap + _re.cap, _re.cap};
    }
    std::vector<edge> edges() {
        int m = int(pos.size());
```

```
39          std::vector<edge> result;
40          for (int i = 0; i < m; i++) {
41              result.push_back(get_edge(i));
42          }
43          return result;
44      }
45      void change_edge(int i, Cap new_cap, Cap new_flow) {
46          int m = int(pos.size());
47          auto& _e = g[pos[i].first][pos[i].second];
48          auto& _re = g[_e.to][_e.rev];
49          _e.cap = new_cap - new_flow;
50          _re.cap = new_flow;
51      }
52
53      Cap flow(int s, int t) {
54          return flow(s, t, std::numeric_limits<Cap>::max());
55      }
56      Cap flow(int s, int t, Cap flow_limit) {
57          std::vector<int> level(_n), iter(_n);
58          std::queue<int> que;
59
60          auto bfs = [&]() {
61              std::fill(level.begin(), level.end(), -1);
62              level[s] = 0;
63              que = {};
64              que.push(s);
65              while (!que.empty()) {
66                  int v = que.front();
67                  que.pop();
68                  for (auto e : g[v]) {
69                      if (e.cap == 0 || level[e.to] >= 0) continue;
70                      level[e.to] = level[v] + 1;
71                      if (e.to == t) return;
72                      que.push(e.to);
73                  }
74              }
75          };
76          auto dfs = [&](auto self, int v, Cap up) {
77              if (v == s) return up;
78              Cap res = 0;
79              int level_v = level[v];
80              for (int& i = iter[v]; i < int(g[v].size()); i++) {
81                  _edge& e = g[v][i];
82                  if (level_v <= level[e.to] || g[e.to][e.rev].cap == 0) continue;
83                  Cap d =
84                      self(self, e.to, std::min(up - res, g[e.to][e.rev].cap));
85                  if (d <= 0) continue;
86                  g[v][i].cap += d;
87                  g[e.to][e.rev].cap -= d;
88                  res += d;
89                  if (res == up) return res;
90              }
91              level[v] = _n;
92              return res;
93          };
94
95          Cap flow = 0;
96          while (flow < flow_limit) {
```

```
            bfs();
            if (level[t] == -1) break;
            std::fill(iter.begin(), iter.end(), 0);
            Cap f = dfs(dfs, t, flow_limit - flow);
            if (!f) break;
            flow += f;
        }
        return flow;
    }

    std::vector<bool> min_cut(int s) {
        std::vector<bool> visited(_n);
        std::queue<int> que;
        que.push(s);
        while (!que.empty()) {
            int p = que.front();
            que.pop();
            visited[p] = true;
            for (auto e : g[p]) {
                if (e.cap && !visited[e.to]) {
                    visited[e.to] = true;
                    que.push(e.to);
                }
            }
        }
        return visited;
    }

  private:
    int _n;
    struct _edge {
        int to, rev;
        Cap cap;
    };
    std::vector<std::pair<int, int>> pos;
    std::vector<std::vector<_edge>> g;
};

#endif  // MAXFLOW_HPP
```

A.2 Генерація випадкових графів та шляхів

```
#ifndef RANDOM_UTILITY_HPP
#define RANDOM_UTILITY_HPP 1

#include "maxflow.h"

#include <random>

template <class RNG>
mf_graph<int> random_graph(int n, int p, RNG& gen) {
    std::uniform_int_distribution<int> p_dist(1, 100), c_dist(1, 100);
    mf_graph<int> graph(n);
    for (int u = 0; u < n; ++u)
        for (int v = 0; v < n; ++v)
            if (u != v && p_dist(gen) <= p)
```

```
                graph.add_edge(u, v, c_dist(gen));
    return graph;
}

#include <algorithm>

template <class RNG>
mf_graph<int> random_variant(mf_graph<int> graph, int r, RNG& gen) {
    std::uniform_int_distribution<int> c_dist(-r, r);
    const auto edges = graph.edges();
    const auto m = edges.size();
    for (int i = 0; i < m; ++i) {
        const auto new_cap = (edges[i].cap * (100 + c_dist(gen))) / 100;
        graph.change_edge(i, std::max(1, std::min(100, new_cap)), 0);
    }
    return graph;
}

mf_graph<int> stable_variant(mf_graph<int> graph, int r) {
    const auto edges = graph.edges();
    const auto m = edges.size();
    for (int i = 0; i < m; ++i) {
        const auto new_cap = (edges[i].cap * (100 - r)) / 100;
        graph.change_edge(i, std::max(1, new_cap), edges[i].flow);
    }
    return graph;
}

mf_graph<int> semi_stable_variant(int n, mf_graph<int> graph, int r) {
    const auto edges = graph.edges();
    const auto m = edges.size();
    for (int i = 0; i < m; ++i) {
        const auto edge = edges[i];
        if (edge.from == 1 || edge.to == n) {
            const auto new_cap = (edge.cap * (100 - r)) / 100;
            graph.change_edge(i, std::max(1, new_cap), edge.flow);
        } else {
            const auto new_cap = (edge.cap * (100 - (9 * r) / 10)) / 100;
            graph.change_edge(i, std::max(1, new_cap), edge.flow);
        }
    }
    return graph;
}

#include <utility>

template <typename T>
using point = std::pair<T, T>;

template <class RNG, typename T>
point<T> random_disk_point(T r, RNG& gen) {
    std::uniform_int_distribution<T> xy_distribution(-r, r);
    T x = 0, y = 0;
    do {
        x = xy_distribution(gen), y = xy_distribution(gen);
    } while (x * x + y * y > r * r);
    return {x, y};
}
```

```

template <typename T>
int sign(T x) {
    return (x > 0) - (x < 0);
}

#include <vector>

template <class RNG, typename T>
std::vector<point<T>> random_disk_path(T r, point<T> s, point<T> t, RNG& gen) {
    std::uniform_int_distribution<int> dir_distribution(0, 1);
    std::vector<point<T>> path = {s};
    while (s != t) {
        T dx = sign(t.first - s.first), dy = sign(t.second - s.second);
        if (!dx) {
            s.second += dy;
        } else if (!dy) {
            s.first += dx;
        } else {
            auto dir = dir_distribution(gen);
            auto v = dir ? point<T>{s.first + dx, s.second} \
                : point<T>{s.first, s.second + dy};
            if (v.first * v.first + v.second * v.second > r * r)
                v = (!dir) ? point<T>{s.first + dx, s.second} \
                    : point<T>{s.first, s.second + dy};
            s = v;
        }
        path.push_back(s);
    }
    return path;
}

#endif // RANDOM_UTILITY_HPP
```

A.3 Моделювання частоти насичення ребер

```
#include "maxflow.h"
#include "random_utility.h"
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

int main() {
    cin.tie(0)->sync_with_stdio(0);
    cout << fixed << setprecision(2);

    int t; cin >> t; while (t--) {
        int n, p;
        cin >> n >> p;

        mt19937 gen(0);
        int trials = 0, flow = 0;
        auto start = high_resolution_clock::now(), now = start;
        vector<vector<int>> cnt(n, vector<int>(n));

        do {
```

```
            auto graph = random_graph(n, p, gen);
            flow += graph.flow(0, n - 1);
            for (auto edge : graph.edges())
                if (edge.cap == edge.flow)
                    ++cnt[edge.from][edge.to];
            now = high_resolution_clock::now();
            ++trials;
        } while (duration_cast<milliseconds>(now - start).count() < 10'000);

        int st_edges = 0, uv_edges = 0;
        for (int u = 0; u < n; ++u)
            for (int v = 0; v < n; ++v) {
                if (u == 0 || v == n - 1)
                    st_edges += cnt[u][v];
                else
                    uv_edges += cnt[u][v];
            }
        cout << trials << " " << flow * 1. / trials << \
         " << st_edges * 1. / trials << " " << uv_edges * 1. / trials <<  "\n";

        for (int u = 0; u < n; ++u) {
            for (int v = 0; v < n; ++v)
                cout << cnt[u][v] * 1. / trials << " ";
            cout << "\n";
        }
        cout << "\n";
    }

    return 0;
}
```

A.4 Порівняння пом'якшеної стратегії

```
#include "maxflow.h"
#include "random_utility.h"
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

int main() {
    cin.tie(0)->sync_with_stdio(0);
    cout << fixed << setprecision(2);

    int t; cin >> t; while (t--) {
        int n, p, r;
        cin >> n >> p >> r;

        mt19937 gen(0);
        int trials = 0, stable_flow = 0, semi_stable_flow = 0;
        auto start = high_resolution_clock::now(), now = start;

        do {
            auto graph = random_graph(n, p, gen);
            auto stable = stable_variant(graph, r);
            const auto stable_flow_inner = stable.flow(0, n - 1);
            auto semi_stable = semi_stable_variant(n, graph, r);
```

```
            const auto semi_stable_flow_inner = semi_stable.flow(0, n - 1);

            auto start_inner = high_resolution_clock::now(), now_inner = start_inner;
            do {
                auto variant = random_variant(graph, r, gen);
                vector<vector<int>> variant_cap(n, vector<int>(n));
                for (auto edge : variant.edges())
                    variant_cap[edge.from][edge.to] = edge.cap;

                bool stable_good = true;
                for (auto edge : stable.edges())
                    stable_good &= edge.flow <= variant_cap[edge.from][edge.to];
                if (stable_good)
                    stable_flow += stable_flow_inner;

                bool semi_stable_good = true;
                for (auto edge : semi_stable.edges())
                    semi_stable_good &= edge.flow <= variant_cap[edge.from][edge.to];
                if (semi_stable_good)
                    semi_stable_flow += semi_stable_flow_inner;

                ++trials;
                now_inner = high_resolution_clock::now();
            } while (duration_cast<milliseconds>(now_inner - start_inner).count() < 100);
            now = high_resolution_clock::now();
        } while (duration_cast<milliseconds>(now - start).count() < 10'000);

        cout << trials << " " << stable_flow * 1. / trials << " " << semi_stable_flow * 1. / trials << "\n";
    }

    return 0;
}
```

A.5 Інтенсивність руху у круглому місті

```
#include "random_utility.h"
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

int main() {
    cin.tie(0)->sync_with_stdio(0);
    cout << fixed << setprecision(3);

    int t; cin >> t; while (t--) {
        int r;
        cin >> r;

        const int m = r + 1 + r;
        const int offset = r;

        mt19937 gen(0);
        int trials = 0;
        auto start = high_resolution_clock::now(), now = start;

        vector<vector<int>> cnt(m, vector<int>(m));
```

```
22
23          do {
24              auto s = random_disk_point(r, gen), t = random_disk_point(r, gen);
25              auto p = random_disk_path(r, s, t, gen);
26              for (auto v : p) {
27                  ++cnt[v.first + offset][v.second + offset];
28              }
29              ++trials;
30              now = high_resolution_clock::now();
31          } while (duration_cast<milliseconds>(now - start).count() < 1000);
32
33          cout << trials << "\n";
34
35          for (int x = -r; x <= r; ++x) {
36              for (int y = -r; y <= r; ++y) {
37                  if (x * x + y * y <= r * r)
38                      cout << cnt[x + offset][y + offset] * 1. / trials << " ";
39                  else
40                      cout << 0. << " ";
41              }
42              cout << "\n";
43          }
44          cout << "\n";
45      }
46
47      return 0;
48  }
```

A.6 Візуалізація інтенсивності руху

```
1   import matplotlib.pyplot as plt
2   import numpy as np
3
4   fig, axs = plt.subplots(4, 3, figsize=(12, 16))
5   plt.subplots_adjust(left=0.125, bottom=0.1, right=0.9, top=0.9, wspace=0.2, hspace=0.2)
6
7   sz = []
8   with open('disk_intensity.in', 'r') as f:
9       t = int(f.readline().strip())
10      assert t == 12
11      for i in range(t):
12          sz.append(int(f.readline().strip()))
13
14  with open('disk_intensity.out', 'r') as f:
15      for i in range(12):
16          trials = int(f.readline().strip())
17          mat = [[-float(val) for val in f.readline().strip().split()] for _ in range(2 * sz[i] + 1)]
18          f.readline()
19
20          axs[i // 3][i % 3].imshow(mat, cmap='PuOr')
21  plt.show();
```