Київський національний університет імені Тараса Шевченка

Факультет комп'ютерних наук та кібернетики

Кафедра теоретичної кібернетики

Кваліфікаційна робота на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

ПОРІВНЯННЯ ЕФЕКТИВНОСТІ АЛГОРИТМІВ РОЗВ'ЯЗАННЯ ЗАДАЧІ КАНАДСЬКОГО ВОДІЯ

Виконав студент 4 курсу

Челноков Семен Ілліч _____

(підпис)

Науковий керівник:

к.ф.-м.н., доц. Ставровський Андрій Борисович _____

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теоретичної кібернетики

«____» _____ 2021 р.,

протокол № ___

Завідувач кафедри Ю. В. Крак

_____

Київ - 2021

РЕФЕРАТ

Обсяг роботи: 45 сторінок, 27 ілюстрацій, 1 таблиця, 5 джерел і 5 додатків.

Ключові слова: ЗАДАЧА КАНАДСЬКОГО ВОДІЯ, ПОШУК НАЙКОРОТШОГО ШЛЯХУ, ПЕРЕКРИТІ РЕБРА, АЛГОРИТМ.

Об'єктом дослідження є задача кандського водія, предметом дослідження є робота алгоритмів для розв’язання задачі канадського водія.

Метою роботи є порівняльний аналіз ефективності алгоритмів для розв'язання задачі кандського водія. В якості засобу для збору даних було обрано середовище програмування Visual Studio 2019 та мову програмування Python 3.7.1. Для роботи зі структурою даних було обрано бібліотеку NetworkX та для роботи зі структурою даних було обрано бібліотеку heapq.

В роботі досліджені основні постановки задачі кандського водія та алгоритми для їх розв’язання.

Були отримані наступні результати: виконано збір даних за допомогою реалізованої програми, що моделює умову задачі канадського водія. Проведено порівняльний аналіз алгоритмів, що показав ефективність створених альтернативних алгоритмів до жадібного та алгоритму «очікування». Дана робота може застосовуватись для ознайомлення з темою та збору інформії про ефективність алгоритмів для вирішення задачі канадського водія. Також реалізована програма може бути вдосконалено та використовуватись, як застосунок для порівняльного аналіз різних алгоритмів та задачах на графах.

ЗМІСТ

ВСТУП

Практично кожного дня ми стикаємся з даною задачою: дійти з точки А в точку Б. Наприклад: ходячи в магазин за продуктами, чи їдучи на таксі, або на власній машині на роботу. Перед нами постає задача вибору оптимального шляху з усіх можливих. Ну і як зазвичай ми уже звикли робити - обирати найшвидший варіант з усіх можливих. Але от не задача, коли на половині шляху виявляється, що на обраному нами шляху дорога перекрита на ремонт чи сталася аварія і рух зупинено. І як же ш тоді обрати оптимальний шлях? З схожою проблемою зіштовхнулись Христос Пападимитриу и Михалис Яннакакис у 1989 і вирішили назвати таку проблему – задача канадського водія (англ. Candian Traveller Problem) [1]. Це узагальнений клас задач про найкротший шлях для графів, які частково видимі. По мірі дослідження графа ми враховуємо вклад ребер, які дослідили, а нові відвідані вершини дають інформацію про невідому частину графа. Таким чином ми враховуємо увесь витрачений час та вартість ребер на пошук оптимального шлях при проході графу, навіть, якщо ми проходили по вершинах, які не входять в найкоротший шлях.

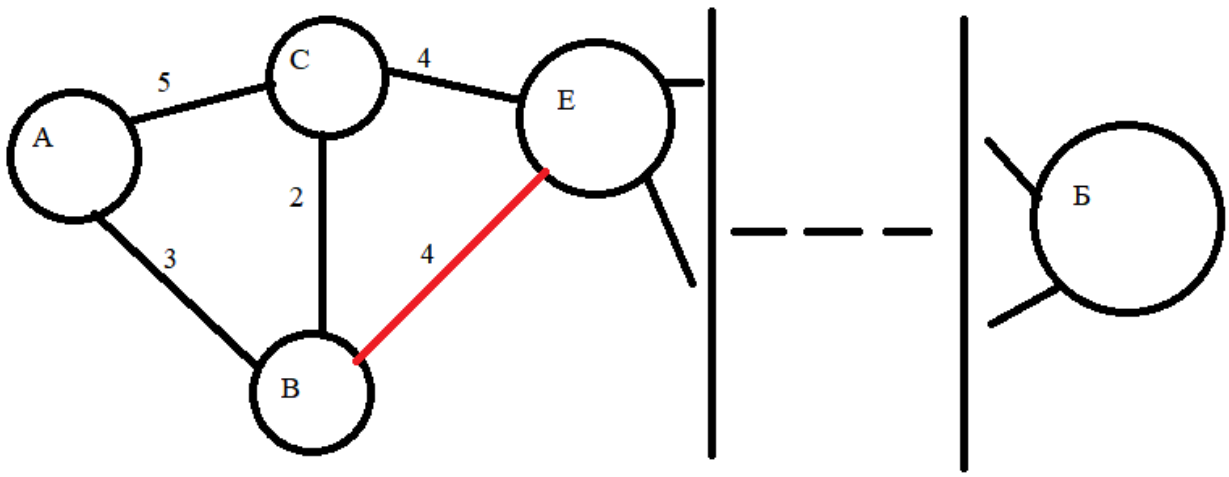


рисунок 1

Дана тема є досить актуальною, оскільки, така задача моделює процеси з якими зустрічаються науковці в робототехніці навіть на сьогоднішній день, це є проблема пошуку оптимального шляху для робота на невідомії місцевості і необхідно придумати стратегію обходу певної поверхні для отримання найкрашого результату, при цьому мінімізувати усі ресурси витрачені на виконання стратегії, що також включає в себе пошук найкоротшого шляху, а це є уже напряму задача канадського водія [8].

Мета моєї роботи є реалізація алгоритмів для вирішення задачі оптимізації, а саме задачі канадського водія та порівняння ефективності їх роботи між собою за часом та знайденим оптимальним шляхом. Методом дослідження є збір даних з комп'ютерної програми, яка буде використовувати певний алгоритм та також аналіз ефективності цих алгоритмів. Згодом після аналізу ефективності можна визначити який з методів є ефективнішим для вирішення задачі та дослідити проблеми, що можуть виникати при виборі того чи іншого алгоритму. Моя робота може виконуватися сумісно з роботами по дослідженню інших алгоритмів для вирішення цієї задачі, що дозволить скороти час на дослідження ефективності того чи іншого алгоритму.

РОЗДІЛ 1 ТЕОРІЯ ГРАФІВ

1.1 ГРАФ

Предметом перших завдань в теорії графов були конфігурації, що існують з точок та ліній, що їх з'єднують. У цих задачах було несуттєво, прямі лінії зв'язку або вони є криволінійними неперервними дугами, що з'єднують дві конкретні точки, де розміщені ці лінії, вони є довгими або короткими. Суттєво лише те, що вони з'єднують дві дані точки. Це призводить до визначення графі як абстрактного математичного поняття.

Граф – це об'єкт, що містить елементи та зв'язки між ними. Зазвичай позначається, як пара елементів G = (V, E), де G – (англ. Graph) відповідно граф, V – (англ. Vertex) вершини, E$\subseteq V * V$ – (англ. Edges) ребра (зв'язки). Множина ребер графу позначають E(G), а множина вершин V(G). m(G)=|E(G)| – кількість ребер графу, n(G)=|V(G)| – кількість вершин графу. Степінь вершини - це кількість ребер, що поєднують цю вершину з іншими.

1.2 ОСНОВНІ ВИДИ ГРАФІВ

Графи поділяють на орієнтовані та неорієнтовані, також бувають змішані. У перших з них всі ребра мають напрямок, а саме початкову і кінцеву вершину. Неорієнтовані, навпаки мають ребра у яких немає напрямку. А змішані відповідно які містять обидва види таких ребер.

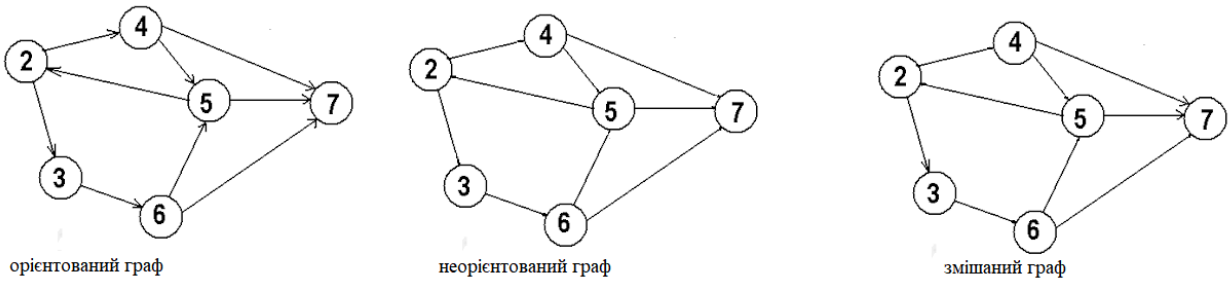


рисунок 2

Також існують, ще одна властивість, яка породжує новий вид графів. Навантажені графи – це граф, у якому кожне ребро має характеристику - вагу, тобто певне число. Такі графи можуть бути як орієнтовані/неорієнтовані чи взагалі змішані, єдина відмінність від усіх інших видів – ребра мають вагу. Це дозволяє задати та вирішувати певний клас задач, наприклад задачу комівояжера, де вагу ребра можна задати, як витрати певного ресурсу палива, часу, грошей(платна дорога) чи тощо. На рис. 6 зображений навантажений граф, де числа на ребрах відображаю вартість проходу по ним.

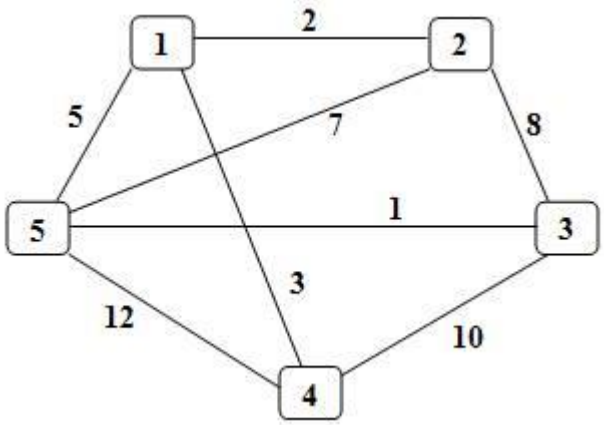


рисунок 3

1.3 СПОСОБИ ЗАДАННЯ ГРАФІВ

В основному більшість звикла представляти графи у вигляді малюнків, де зображені вершини, ребра та іноді ваги ребер(якщо вони потрібні), такий спосіб зазвичай називають *геометричний* чи *графічний*. Але такий спосіб задання не завжди зручний та іноді навіть важкий для сприйняття. Наприклад, потрібно вирішити задачу комівояжера для великої кількості міст, припустимо тридцять. Навіть, якщо ми зобразимо всю цю інформацію на малюнку, більшість з неї ми загубимо чи переплутаємо у процесі вирішення задачі, якщо деякі дороги перетинаються між собою чи тощо. Також з настанням інформаційної ери(ери комп'ютерів), коли кількість інформації виросла в рази, але актуальність графів/структур не здає свої позиції, то настала потреба в нових способах

задання графів. Такі способи стали би зручними для зчитування комп'ютерною програмою та написання алгоритмів вирішення цих задач. Що давало неабиякі переваги у швидкості та якості виконання в порівнянні з людиною. Для кращого сприйняття і порівняння, представимо один і той самий граф(зорбражений на рис. 7) різними способами:

1. Графічний

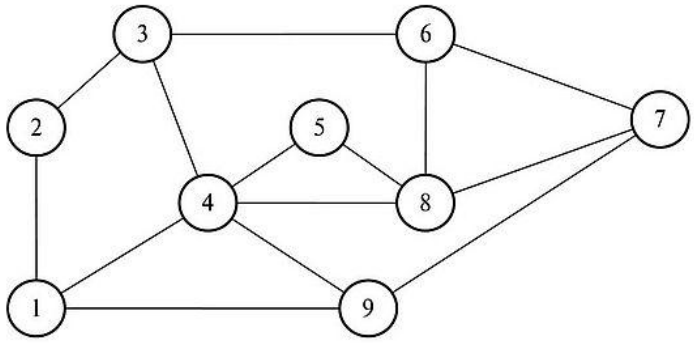


рисунок 4

2. Аналітичний(перелік елементів). Просто перераховуємо всі наявні елементи в множинах вершин та ребер.

   G=(V,E)

   V ={1,2,3,4,5,6,7,8,9},

   E={$e_1 = (1,2)$, $e_2 = (1,4)$, $e_3 = (1,9)$, $e_4 = (2,3)$, $e_5 = (3,4)$, $e_6 = (3,6)$, $e_7 = (6,8)$, $e_8 = (6,7)$, $e_9 = (7,8)$, $e_{10} = (7,9)$, $e_{11} = (8,5)$, $e_{12} = (8,4)$, $e_{13} = (5,4)$, $e_{14} = (4,9)$}

3. Матриця інцидентності. Рядки матриці – вершини, стовпці – ребра. Якщо вершина v_1інцидентна ребру e_1, то елемент матриці А, a_{11} позначаємо 1, якщо ні – 0.

	V\E	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11	e12	e13	e14
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
	2	1	0	0	1	0	0	0	0	0	0	0	0	0	0
	3	0	0	0	1	1	1	0	0	0	0	0	0	0	0
	4	0	1	0	0	1	0	0	0	0	0	0	1	1	1
A=	5	0	0	0	0	0	0	0	0	0	0	1	0	1	0
	6	0	0	0	0	0	1	1	1	0	1	0	0	0	0
	7	0	0	0	0	0	0	0	1	1	0	0	0	0	0
	8	0	0	0	0	0	0	1	0	1	0	1	1	0	0
	9	0	0	1	0	0	0	0	0	0	1	0	0	0	1

рисунок 5

4. Матриця суміжностей. Іноді називають матрицею сусідства. Квадратна матриця порядку |V|, де рядки і стовпчики – вершини графа, а елемент відповідно позначає ребро, яке їх з'єднує. $a_{11} = 0, a_{12} = 1$ ...

	V\V	1	2	3	4	5	6	7	8	9
	1	0	1	0	1	0	0	0	0	1
	2	1	0	1	0	0	0	0	0	0
	3	0	1	0	1	0	1	0	0	0
	4	1	0	1	0	1	0	0	1	1
A=	5	0	0	0	1	0	0	0	1	0
	6	0	0	1	0	0	0	1	1	0
	7	0	0	0	0	0	1	0	1	1
	8	0	0	0	1	1	1	1	0	0
	9	1	0	0	1	0	0	1	0	0

рисунок 6

РОЗДІЛ 2 ЗАДАЧА КАНАДСЬКОГО МАНДРІВНИКА

2.1 ЗАДАЧА ПРО НАЙКОРОТШИЙ ШЛЯХ

В теорії графів велику увагу приділяють задачі про найкортший шлях(англ. shortest path problem) і оскільки, вона на пряму пов'язана з темою моєї роботи, то варто її розглянути.

Задача полягає в наступному: нам дано зважений граф і нас просять знайти оптимальний шлях з стартовий вершини в кінцеву так, щоб ціна яку ми заплатимо, пройшовши цей шлях, була мінімальною. Якщо ж ребра не мають ваги, то задача зводиться до пошуку шляху, що містить мінімальну кількість ребер.

Уточнимо цю неформально поставлену задачу. Є зважений граф G = (V, E) та вершини в ньому v_{start} і v_{end}, а також вагова функція $f: E \to R$, що відображає ребра на їх дійсну вагу. Потрібно знайти послідовність P = $(v_1, v_2, \dots, v_n)$ – шлях у графі, в якому вершини v_i і v_{i+1} суміжні для і ∈ $[1, n)$, $v_1 = v_{start}$, $v_n = v_{end}$, і який має мінімальне значення суми $\sum_{i=1}^{n-1} f(e_{i,i+1})$.

2.2 ВИДИ ПОСТАНОВОК ЗАДАЧІ КАНАДСЬКОГО ВОДІЯ

Одна з найвідоміших задач в теорії графів є задача про пошук найкоротшого шляху в графі, та технології розвиваються і створюються нові проблеми, що призводить до виникнення нових задач та їх стратегій вирішення. Так і сталося з задачею про найкоротший шлях, що призвело до виникнення нових весрій цієї задачі. Це було необхідно для вирішення значної кількості проблем, зокрема для пересування робота по поверхні, що має перешкоди, або ж вони утворюються з часом. Можна припустити, що дані про поверхність(або ж як прийнято називати в робототехніці – сцени) поступають роботу з часом, але як тоді прийняти рішення в якому напрямку продовжувати рух далі?

2.3 ЗАДАЧА ПРО НАЙКОРОТШИЙ ШЛЯХ БЕЗ КАРТИ

Припустимо, ми маємо граф зображений на рис. 10, що має ширину два і поставлена задача знайти шлях з стартової вершини “s” в останню “t”.

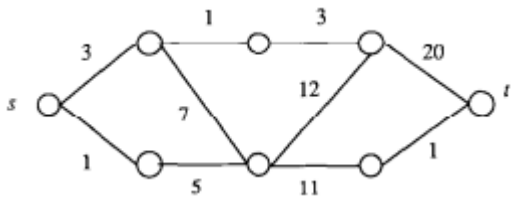


рисунок 7

Зазвичай ми б використовували алгоритми Дейкстра або Беллммана-Форда, та уявімо, що відомості про граф нам поступають ітераційно з якимось певним проміжком часу, або в нас є поле зору, що не дозволяє бачити інші вершини, які не є суміжними з вершиною в якій ми перебуваємо, тоді б отримали підграф зображений на рис. 11.

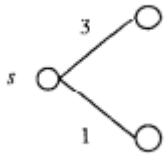


рисунок 8

Як тоді приймати рішеня? Нехай, ми керуємся «раціональним вибором», кожного разу обирти шлях з мінімальною ціною, тоді ми наївно б перейшли по нижньому ребру(рис. 11). І мали би наступний «шматок» графу рис. 12, отримавши такі дані, тепер стає невідомим який з шляхів краще обрати.

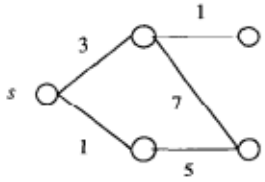


рисунок 9

Також є можливість повернутися назад і інформація яку ми отримали раніше нікуди не зникає, проте загальна вартість завжди враховуватиме ціни ребер, які ми дослідили, навіть, якщо вони не входять до «найкоротшого шляху», який ми знайдемо. Звісно немає жодного способу гарантувати, що ми знайдемо найкоротший шлях (припускаючи, що граф не змінюється з часом і «ніби» був відомий нам заздалегідь). Тоді ми припускаємо, що нашою метою є розробка плану з урахуванням загальної відстані, що ми пройшли має найліпше відношення до найкоротшого шляху. Це і є загальне пояснення проблеми канадського мандрівника. Проте є і інші поставки цієї задачі, які можуть моделювати не менш цікаві процеси та ситуації з якими ми зустрічаємся кожного дня.

2.4 СТОХАСТИЧНА ЗАДАЧА КАНАДСЬКОГО ВОДІЯ

Розглянемо граф зображений на рис. 13 як ми бачимо в графі є ребра, що мають ціну та знаки питання - так ми позначаємо ребра, які можуть виявитись непридатними для проїзду (наприклад аварія, снігопад, тощо) і ця подія нам відображається лише тоді, коли досягнуто сусідню вершину з цим ребром. Також ми отримуємо значення вірогідностей, що вказують відсоток того, що ці дороги можуть бути закриті, або навпаки відкриті. Ця задача чимось схожа, на задачу про найкоротший шлях без карти, проте тут зовсім інша умова. Нам задано увесь граф, в якому нам відомі усі вершина та ребра, а також відповідно ймоірності цих ребер належити реалізації (бути відритими чи закритими), при чому ребра незалежні між собою. Тобто, закриття/відкриття одного ребра не впливає на інше.

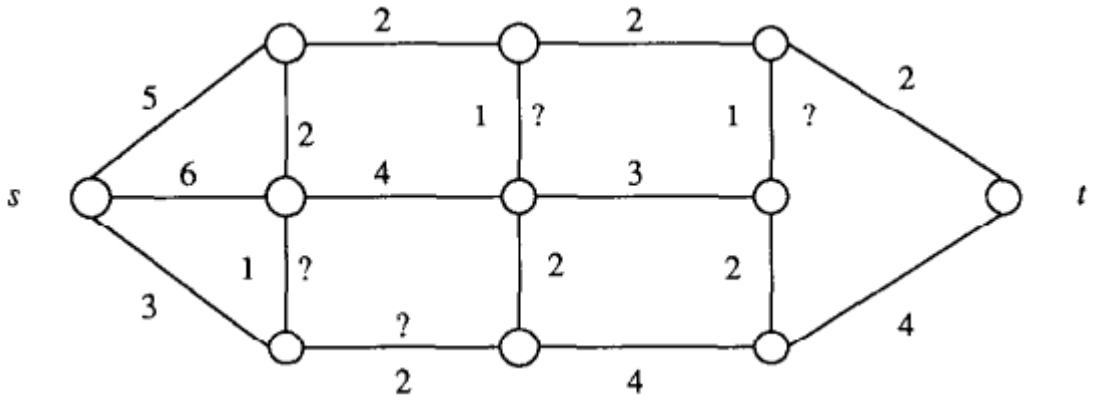


рисунок 10

Якщо дивитися «зверху» на цю задачу, то маємо таку картину: мандрівник обирає собі ребра по яким ходить (враховується усі проходи по ребрам в загальну вартість), а сам граф ніби завмирає і ми як спостерігачі знаємо, які дороги закриті, а які відкриті, що невідомо для нашого мандрівника і він отримує цю інформацію, лише коли опиниться у інцидентній вершині.

2.5 АЛЬТЕРНАТИВНА ВЕРСІЯ СТОХАСТИЧНОЇ ЗАДАЧІ КАНАДСЬКОГО ВОДІЯ

Спрощена версія стохастичної задачі канадського водія, адже вона має меншу складність від альтернативної та легша в реалізації. Умова приблизно така сама, як і в стохастичній версій, окрім того, що нам не заданий розподіл ймовірностей по ребрах графу, проте дано відсоток перекритих ребер від загальної кількості ребер. Так само, мандрівник не знає які саме дороги перекриті, а інформацію отримує лише при досягнені вершини, яка належить перекритому ребру.

2.6 ІНШІ ВЕРСІЇ ЗАДАЧІ ТА ЧИМ ВОНИ ВІДРІЗНЯЮТЬСЯ

В основному в цьому розділі було розглянуто лише стохастичні версії даної задачі, проте це один лише з параметрів, по яким прийнято відрізняти ці задачі. Тому в основному виділяють декілька:

- Як формувати вартість проходу по графу. Адже, наприклад в стохастичній версії метою є мінімізація вартості ціни проходу, яка підраховує ребра в яких ми були та кількість разів, скільки ми їх проходили. А саме для задачі канадського водія метою є мінімізація відношення даного проходу, до найкоротшого шляху.
- Як граф змінюється з часом. В розглянутих мною версіях граф ніяк не змінюється з проходом мандрівника по ребрах, тобто, коли ми дивимся як спостерігачі «зверху», то нам все відомо і все залишається фіксованим. Проте виділяють і інші версії, наприклад коли реалізації графу змінюються з часом і нам заданий розподіл цих реалізацій, проте тут уже буде інша мета задачі, адже важко сказати, що є найкоротшим шляхом в цьому графі.
- Як нові дані надходять до мандрівника. В попередньо представлених версіях дані надходили ітераційно або при досягненні вершин, проте недвано було представлено новий варіант задачі, з кожної вершини можна зробити збір інформації про граф за певну вартість. Тоді б в цьому варіанті метою була б мінімізація вартості проходу та вартості операцій зі сбору інформації

РОЗДІЛ 3 АЛГОРИТМИ

3.1 СТРАТЕГІЯ ТА АЛГОРИТМИ

В цьому розділі наведені алгоритми використані мною під час дослідження. Я обрав саме ці алгоритми, оскільки, вони пов'язані напряму з вирішення даної задачі, а саме з пошуку найкоротшого шляху. Також мною було обрано альтернативний варіант стохастичної задачі в якому ми наперед не знаємо, які ребра графа будуть непридатними для користування, проте нам відомий увесь граф, а саме вершини, ребра та їх вартості. Після цього ми можемо умовно поділити вирішення задачі на дві частини:

1. Вибір наступного кроку. Під цим маю на увазі – яку вершину обрати наступну на нашому шляху.
2. Розробка стратегії. Яким чином приймати рішення в той момент коли ми зіштовхнулись з тим, що обране нами ребро закрите.

Таким чином задача поділилась на класичну задачу пошуку найкорошого шляху, яку можна вирішувати класичними алгоритмами, тобто, проаналізувати весь граф та знайти найкоротший шлях і йти по ньому до поки ми не прийдемо в кінцеву вершину, або потрапимо на дорогу, що є закритою. Далі необхідно розробити стратегію та слідувати строго їй, наприклад, якщо на обраному шляху маємо закриту дорогу, тоді робти крок назад, передавати алгоритму нову інформацію та отримувати новий шлях і так по колу, таким чином ми або прийдемо в кінцеву вершину, або обійдемо всі вершини і отримаємо, що кінцева точка недоступна(до неї всі дороги перекриті).

Я зупинився на досить простій стратегій, вона звучить так – спочатку знаходимо найкоротший шлях з усього графу та вважаємо, що всі ребра доступні для проходу, далі обраним алгоритмом проходимо по цьому шляху, до поки не дійдемо до кінцевої вершини, або якщо ми потрапили на перекриту

дорогу, то видаляємо її з графа, та знов шукаємо алгоритмом найкоротший шлях уже з даної точки до кінцевої. Таким чином ми охопили усі можливі варіанти, ми або дійдемо до кінцевої точки, або обійдемо весь доступний граф і отримаємо, що кінцева вершина недоступна.

3.2 ЖАДІБНИЙ АЛГОРИТМ

Жадібний алгоритм – алгоритм, робота якого полягає у виборі найкращого варіанту на кожній ітерації в припущені, що кінцевий варіант буде також найкращим. Цей алгоритм дуже спецфічний, тому розглянимо його переваги та недоліки.

Переваги:

- Швидкий в реалізації та написанні. Оскільки на кожному кроці потрібно робити лише перевірку на оптимальний варіант.
- Ефективний за часом виконання. Через те, що швідкість алгоритму достатньо легко передбачити. Частіше за все складність лінійна, тобто, час виконання залежить лінійно від кількості вхідних даних.

Недоліки:

- Не завжди є можливість його використовувати. Наприклад, для задачі комівояжера та мінімального розфарбовування графа, цей алгоритм не дає оптимальних рішень. Тому його ефективно використовувати лише для певного класу задачу.
- Не завжди дає оптимальні рішення. Може бути реалізований та працювати, але неефективно, оскільки, не даватиме оптимальне вирішення задачі в порівнянні з іншими алгоритмами.

При вирішенні задачі було розроблено два види алгоритму, для двох різних постановок задач. Перший досить банальний ідемо по отриманому нами

найкоротшого шляху і далі враховуємо вартості пройдених нами шляхів, якщо зустрічаємо закрите ребро, то видаляємо його та заново будуємо шляї.

Другий варіант, це для цієї самої задачі, проте у мандрівника з'явилась можливість «огляду» - це означає, що коли мандрівник знаходиться в вершині він може продивитись усі інцидентні ребра до цієї вершини та одразу помітити собі ті маршрути, які є закритими – наприклад видалити їх з графу. Тоді ми матимемо більше інформації і це може вплинути на швидкість пошуку нового шляху, адже якщо ми попадемо на дорогу, яка належить нашому найкоротшому шляху, та вона є перекритою, то ми знатимемо усі перекриті дороги(ребра), які інцидентні вершинам в яких ми бували. Для порівняння можна подивитись на рис. 14. Необхідно почати шлях з вершини «0» та прийти в вершину «9». Обведені вершини це обраний шлях нашим алгоритмом, римські цифри – це номер ітерації(одна ітерація – один прохід перебру, далі іде наступний прохід по циклу). Обидва рази ми на третій ітерації зіштовхуємся з тим, що найкоротший шлях перекрито, тому необхідно видалити це ребро та побудувати новий шлях. І тут ми отримуємо різницю, адже на другій ітерації в першому алгоритмі ми просто пройшли далі по зданому шляху, а в другому варіанті алгоритму ми видалили ці ребра. Отже, використовуючи другий варіант, ми матимемо перевагу, адже ми не витратимо час пробуючи пройти в вершину «2» чи повернувшись назад в вершину «7» не будемо пробувати ідти в «4» та «1» вершину, а одразу побудуємо інший шлях, що може вплинути, як на вартість усього шляху так і на час пошуку загалому.

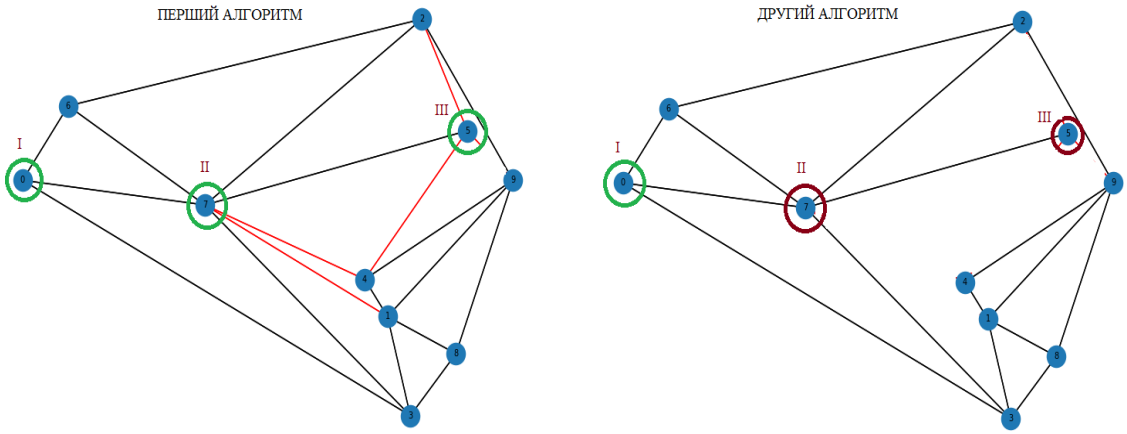


рисунок 11

3.3 АЛГОРИТМ ДЕЙКСТРИ

Це один з найвідоміших алгоритмів пошуку найкоротшого шлях, винайдений математиком Едсгером Дейкстром у 1959 році. Метою алгоритму є пошук мінімального шляху з початкової вершини в усі інші вершини, що належать графу.

На першому етапі необхідно промаркувати усі вершини позначками нескінченності, а стартуву нулем, адже ми з неї починаємо. Так робимо, оскільки вважаємо, до поки ми не перейшли в цю вершину і не знаєм вагу ребра, то будем вважати відстань дуже великою (наприклад нескінченність, або програмно можна позначити, як дуже велике число (2^31-1)) це нам знадобиться для подальших перевірок.

Наступним етапом буде прохід по всіх сусідніх вершинах та перевірка: чи більша поточна вартість проходу до вершини ніж та, що ми витратили щойно. Якщо так – то змінюємо віповідно на меншу, якщо ні – залишаємо маркер таким самим і ідемо до наступної сусідньої вершини. Такі операції проводимо з кожною сусідньої. Коли ми пройшлих всіх сусідів викреслюємо вершину в якій ми знаходимся та переходимо до наступного «шару». При чому далі, потрібно враховувати вартості ребер зі стартової вершини. Отже, в кінці ми отримаємо мітки для кожної вершини, а саме мінімальні вартості проходу в дану вершину з

початкової (мінімальні, оскільки ми кожно разу перевіряли чи поточний шлях дешевший ніж той, що вже є і переприсвоювали). Алгоритм завершує свою роботу тоді, коли всі вершини викреслено. Покрокову дію алгоритму можна побачити на рис. 15.

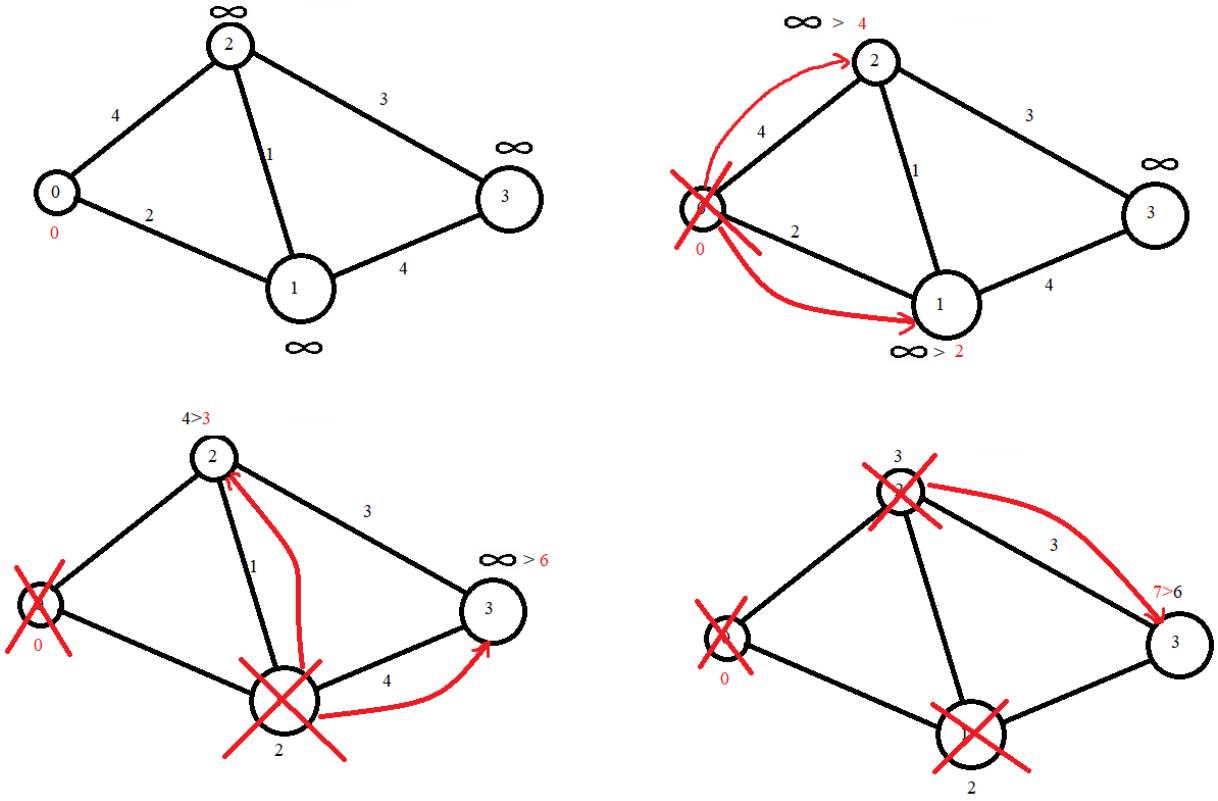


рисунок 12

І пілся роботи алгоритми ми маємо граф, де біля кожної вершини проставлений маркер з мінімальною вартістью зображений на рис. 16. Також можна додати маркер «попередньої вершини», що дозволить будувати найкоротші шляхи.

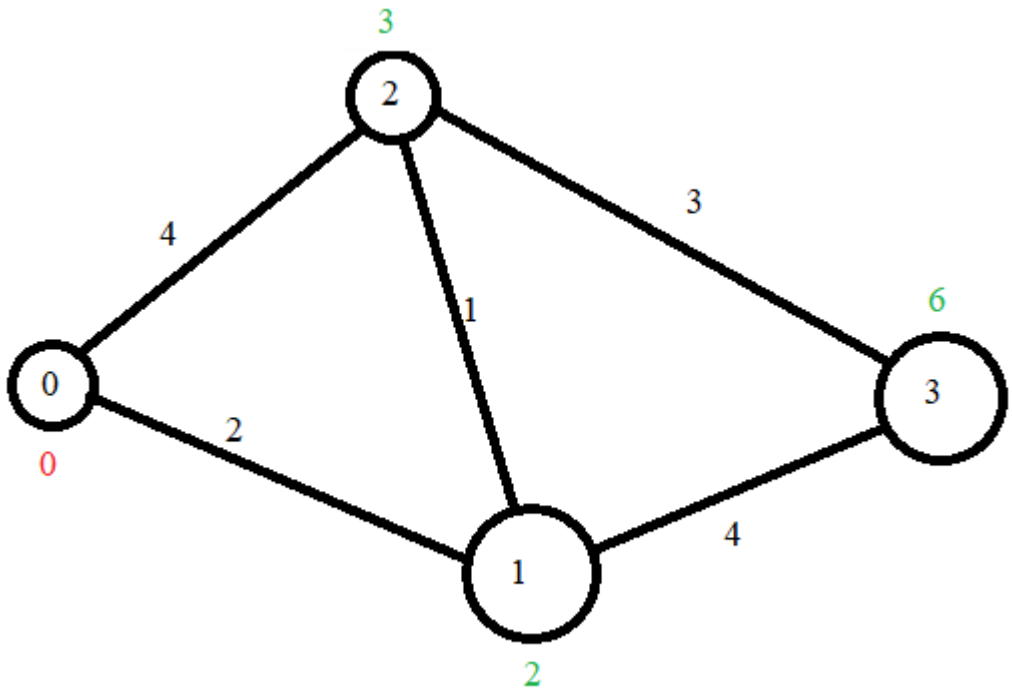


рисунок 13

3.4 АЛГОРИТМ «ОЧІКУВАННЯ»

Цей алгоритм був створений під час дослідження теми моєї роботи. Я вирішив подивитись на задачу з іншого боку та більше наблизити її до реальності. Наприклад, в країнах Європи досить розповсюджене таке явище, як платні дороги. Тому було вирішено це також додати в цю задачу та подивитись, як це вплине на ефективність алгоритмів.

Задача залишається такою самою, якийсь відсоток доріг буде перекритий, ми не знаємо який і які саме ребра, і нам необхідно дістатись з стартової вершину в кінцеву. Проте ми додаме таке поняття, як штраф – це випадкова велечина, яка не перевищує вагу ребра, та є не нульовою. Суть полягає в тому, щоб коли ми прийшли в вершину, шлях з якої перекритий, ми мали можливість пройти по цьому ребру, проте заплативши певну ціну або шукати інший шлях.

Також було створено аналогічний варіант до жадібного алгоритму. Під час проходу по шляху в кожній вершині ми переглядаємо всі інциденті ребра та

зберігаємо усю інформацію про перекриті дороги та вартість їх проходу включаючи штраф. Тоді у нас є більше простору для дій: коли ми стикаємось з першою перекритою дорогою на обраному нами шляху, знаючи інформацію, про всі ребра, які інцидентні вершинам, що ми пройшли, ми можемо заново зробити пошук найкоротшого шляху, таким чином зменшити загальну вартість шляху, проте це може зайняти більше часу.

РОЗДІЛ 4 РОЗРОБКА ПРОГРАМИ

Дану програму було розроблено в середовищі програмування visual studio 2019 на мові програмування Python 3.7 з використанням таких пакетів: NumPy, math, random matplotlib, SciPy, heapq, networkx, xlswriter. Такі пакети досить зручні для математичних операцій та розробки алгоритмів.

4.1 РОЗРОБКА ІНТЕРФЕЙСУ

Інтерфейс користувача на рис. 17 було розроблено за допомогою програми Qt Designer. Інтерфейс використовувася для зручного збору даних та відображення роботи алгоритмів. На далі може використовуватись як повноцінний додаток для перевірки роботи інших алгоритмів, протей ого необхідно вдосконалити.

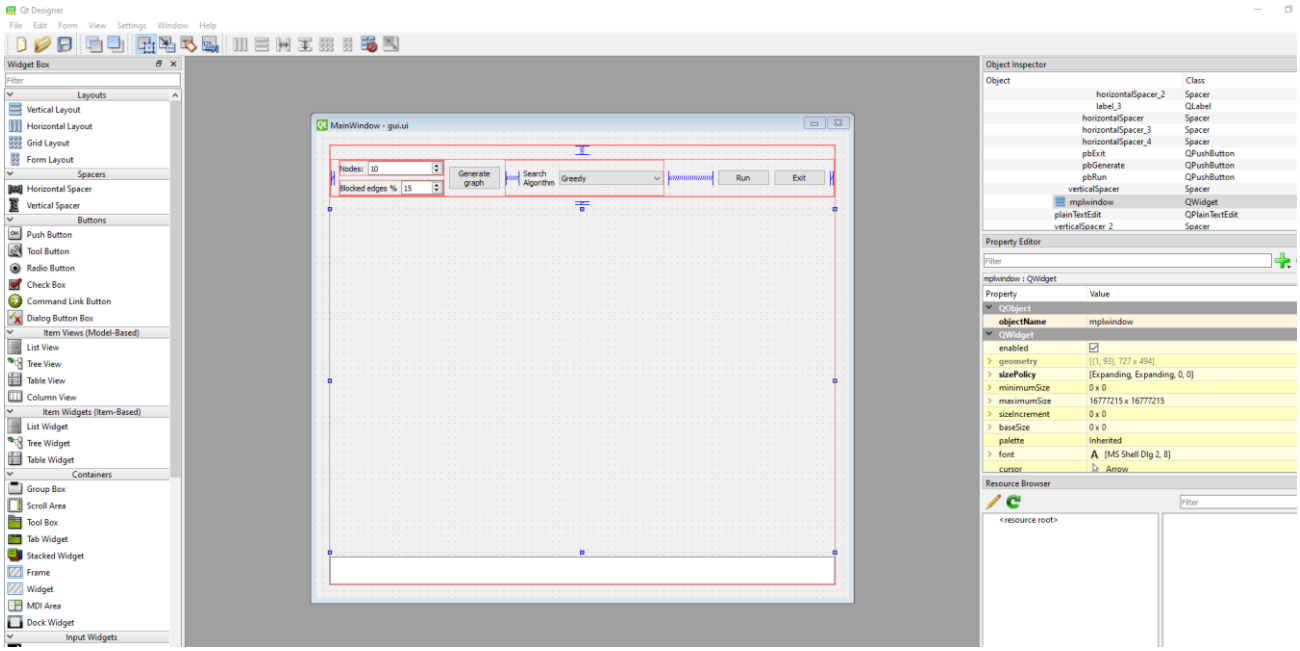


рисунок 14

Використано такі елементи для побудови інтерфейсу:

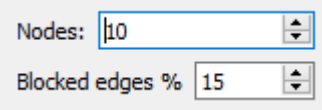


рисунок 15

- QSpinBox – для вибору кількості вершин графу та відсотку блокованих доріг.

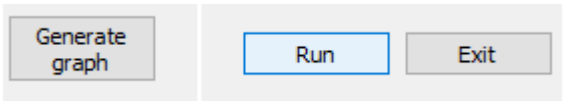


рисунок 16

- QPushButton – для виконання таких процесів як генерація графа, виконання алгоритму та виходу з програми.

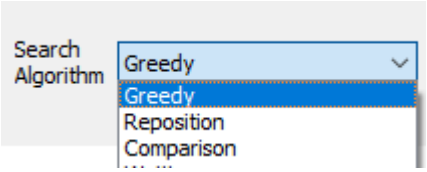


рисунок 17

- QComboBox – для вибору алгоритму, що буде виконуватись.

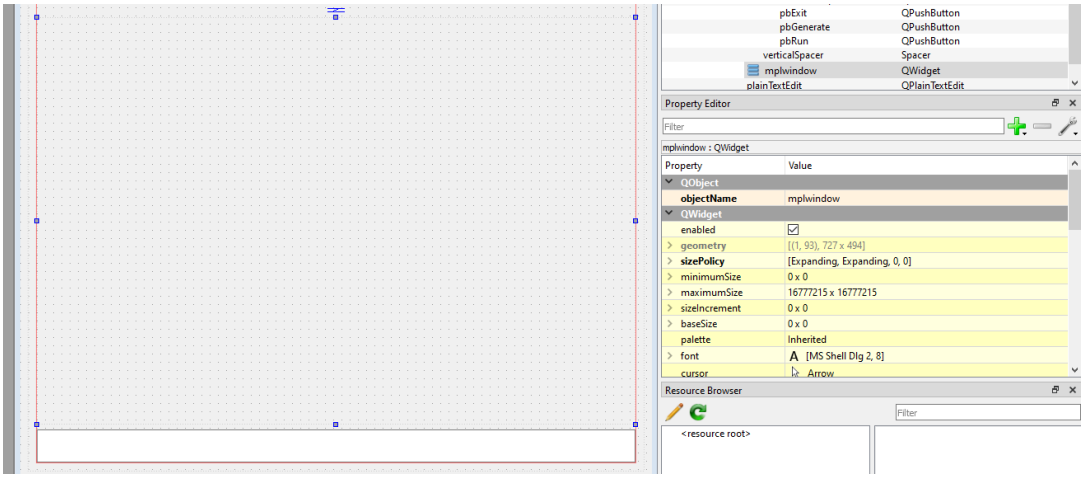


рисунок 18

- QWidget – для виводу інформації.
- QLabel – для підпису описаних вище елементів, що полегшують орієнтування користувача.
- HorizontalSpacers та VerticalSpacers – для правильної “геометрії” та нормального відображення усіх елементів під час роботи програми.

Так, пілся розробки інтерфейсу зберігаємо його з роширенням *.ui. Згодом у python- файлі, який буде виконуватись, імпортуємо необхідні бібліотеки PyQt5 та за допомогою команди uic.loadUi(‘gui.ui’) завантажуємо наш інтерфейс.

І в кінці ми маємо такий користувацький інтерфейс:

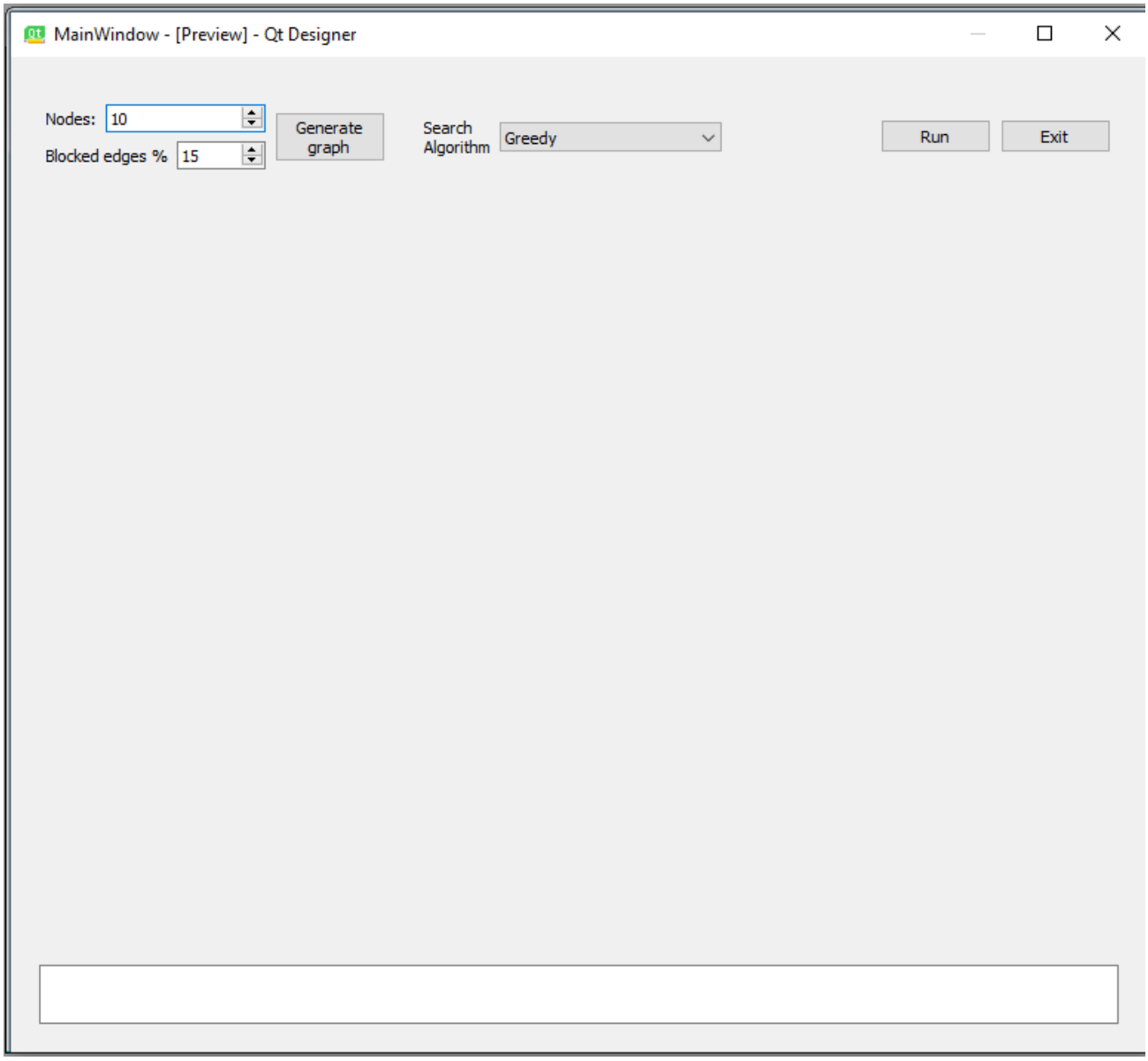


рисунок 19

4.2 ПОБУДОВА ГРАФА

Оскільки мною не було знайдено готових даних для цього дослідження, було прийнято рішення написати модуль, який буде генерувати граф для вирішення задачі. Для генерації графу необхідно вказати кількість вершин, яка буде в графі і вказати відсоток заблокованих доріг.

Модуль розбитий на дві частини, в першій ми будуємо граф, отримавши вказану кількість вершин, яку вводимо в інтерфейсі, а в другій моделюємо процес блокування доріг з відсотком, який ми вказали в інтерфейсі. Таким чином це дає змогу моделювати задачу на різній кількості вершин та кількості ребер, що є заблокованими. Та це дає можливість моделювати велику кількість процесів, що наближені до реальності.

Перша частина складається з однієї функції, що приймає об'єкт створений цим модулем та кількість вершин. Далі створюється список вершин. В якому одразу додаємо значення першої точки (0; 22) – це майбутні координати точки. Згодом генеруємо координати усіх інших точок, окрім останньої, за допомогою модуля random в межах від [3,40]. І після циклу додаємо останню точку з координатами – (43, 22). Таким чином ми полегшели собі побудову графу, адже перша і остання вершина стоять на одній вертикалі і стоять в різних «кутках» вікна в якому відображається граф для користувача. А такі межі координат було взято з розміру вікна, що зображує граф у інтерфейсі. Згодом було необхідно зробити триангуляцію Делоне це необхідно для коректного відображення та «правильної» побудови ребер, щоб вони не накладались одне на одного і було нормально їх бачити. Таким чином ми заповнюємо наш пустий граф трикутниками так, що кожна нова вершина дає додаткові два ребра, якщо через одну трійку вершин можно описати коло і цьому колу будуть належати тільки ці три вершини – це і є правило триангуляції Делоне. Приклад триангуляції

можна побачити на рис. 23. Модуль, що реалізує описані дії, наведений у **додатку А.**

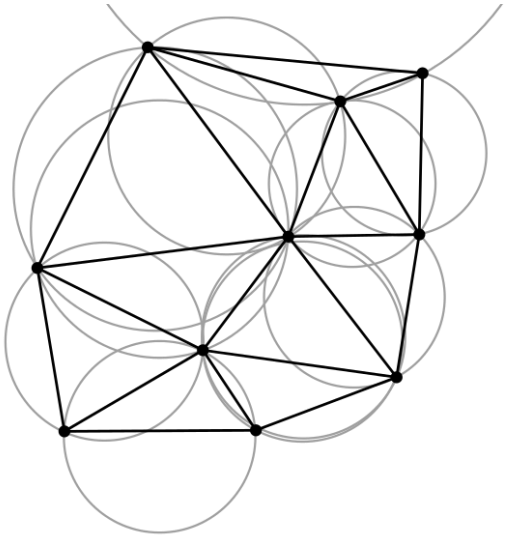


рисунок 20

Створюємо об'єкт класу networkx, що є графом. Та за допомогою вбудованих функцій додаємо ребра і виставляємо відповідні атрибути: колір(чорний), вагу дорівнює довжині вектору між точками, та штраф(випадкова велечина від 5 до 10). І аналогічно для вершин створюємо словник, який містить назву вершини - як ключ, а позицію(координати) – як значення та виставляємо атрибути для цієї вершини, який зберігатиме цей щоденник. І згодом зберігаємо і повертаємо цей об'єкт.

4.3 ПОБУДОВА ЗАБЛОКОВАНИХ ДОРІГ

З інтерфесу ми підтягуємо відсоток ребер, які мають бути закриті, тому використаємо функцію, якій на вхід передаватимемо об'єкт – граф та значення – відсоток перекритих ребер. За допомогой методів бібліотеки networkx можемо виставити властивості для ребер, таким чином додаємо властивість «color» і виставляємо значення «black» для усіх ребер. Знаходимо кількість ребер, яку необхідно позначити закритими, округлюючи це число завжди до більшого та задопомогою функції random.sample() одразу отримуємо список заблокованих

доріг і наступною командою проходимся по цьому списку та міняємо колір кожного заблокованого ребра на червоний. І повертаємо назад цей об'єкт. Функція, що реалізує описані дії, наведена у **додатку Б.**

4.5 РЕАЛІЗАЦІЯ АЛГОРИТМУ ДЕЙКСТРИ

Алгоритми було реалізовано, як окремий модуль, який містить три функції: пошук найкоротшого шляху, знаходження загальної вартості шляху та видача знайденого шляху. Основною є функція найкоротшого шляху, яка приймає на вхід сам граф, стартову та кінцеву вершину. За допомогою циклу ми обходимо усі вершини графу та відповідно залишаємо певні почзначки використовуючи методи бібліотеки nx, а саме присвоюємо вершинам такі властивості:

- попередня вершина з якої ми прийшли;
- чи бували ми в цій вершині;
- поточна довжина шлях з початкової вершини;

Та за допомогою бібліотеки heapq використовуємо таку структуру даних, як купи, що дозволяє зберігати дані з пріоритетами. Таким чином, ми в купі тримаємо ключі у вигляді вершин та значення – мінімальний шлях до цієї вершини, а якщо нам необхідно звернутись до якоїсь вершини з купи, то ми отримаємо саме мінімальне значення, що полегщує роботу нашої програми.

За допомогою функції «видачі найкоротшого шляху» ми можемо знайти шлях, адже пілся роботи алгоритму на кожній вершині ми маємо властивість – попередня вершина з якої ми прийшли, то ми просто ідемо з кінцевої вершини в початкову і потім використовуючи метод списків reverse() розвертаємо наш шлях і отримуємо найкоротший шлях, який нам потрібний. Описані вище дії реалізовані у модулі, який наведений у **додатку В.**

Також було реалізовано за допомогою цього алгоритму пошук оптимального шляху уже в умовах задачі, де можуть бути перекриті ребра. Оскільки уже було реалізовано окремий модуль з алгоритмом Дейкстри, то залишається прибрати з графу перекриті ребра та виконати алгоритм Дейкстри. Оскільки, доведено, що алгоритм Дейкстри знаходить найкоротший шлях в графі, то видаливши перекриті ребра з графу ми просто перейшли до задачі найкоротшого шляху. Даний модуль було реалізовано для порівняння ефективності виконання інших алгоритмів і також, щоб побачити, як перекриття доріг впливає на вибір шляху.

4.6 РЕАЛІЗАЦІЯ ЖАДІБНОГО АЛГОРИТМУ

Даний алгоритм використовується для виконання стратегії. Реалізований таким чином:

1. Засікаємо таймер.
2. Ми викликаємо функцію модуля Дейкстри, що повертає нам найкоротший шлях.
3. В циклі йдемо по отриманому шляху.
   3.1. На кожній вершині робимо перевірку обраного нами ребра, а саме – чи є воно перекритим.
   3.2. якщо ні – додаємо його вагу у загальну вартість та додаємо його у пройдений нами шлях.
4. якщо так – видаляємо перекрите ребро з графу та викликаємо функцію найкоротшого шляху і переходимо до пункту 2.
5. Вимикаємо таймер.

В кінці викликаємо функцію пошуку загальної вартості шляху передаючи їй шлях, по якому ми йшли. І користувачу відповідно повертаємо пройдений шлях, загальну вартість та час, який ми витратили на прохід від початкової вершини до кінцевої. Реалізація даного алгоритму наведена у **додатку Г.**

Також було реалізовано альтернативний варіант даного алгоритму, відмінність полягає в тому, що перед перевіркою(пункт 4) ми перевіряємо усі інцидентні ребра вершини в якій ми знаходимся і якщо є перекриті, то ми їх одразу видаляємо. А замість перевріки чи є ребро перекритим, ми перевіряємо чи взагалі існує таке ребро.

4.7 РЕАЛІЗАЦІЯ АЛГОРИТМУ «ОЧІКУВАННЯ»

В даном алгоритмі застосовується таке поняття, як штрафи («penalty»), що впливає на умову задачі, а саме додає до заблокованих ребер цю властивість. І таким чином ми перевіряємо – «а щоб було, якщо можна заплатити додаткову вартість для проходу по данному ребру і як би це вплинуло на загальну вартість шляху та час пошуку цього шляху?». Даний алгоритм, дещо змінює стратегію задачі. Алгоритм реалізований за схожим принципом з жадібним алгоритмом. Ми ідемо по знайденому найкоротшому шляху, що надав нам алгоритм Дейкстри, в циклі робимо перевірку ребра, яке знаходиться наступним у списку шляху, якщо воно є перекритим –зберігаємо в окрему змінну «penalty» його штраф та додаємо це ребро до шляху, яким ми ідемо, якщо воно відкрите – просто записуємо його в наш шлях. Аналогічно до жадібного засікаємо час з моменту виклику функції найкоротшого шляху і фіксуємо час виходу з циклу. А за пройденим шляхом викликаємо функцію пошуку вартості шляху і до отриманого значення додаємо значення змінної «penalty».

Подібно до жадібного алгоритму, реалізовано альтернативу до цього алгоритму, коли ми уже знаходимось в середині циклу, в змінну «restart» записуємо значення ребра, яке є наступним на нашому шляху і потім перевіряємо всі інцидентні ребра, проте ми їх не видаляємо, а змінюємо їх маркер, щоб вони значилися відкритими (включаючи ребро, яке є наступним на шляху, якщо воно є перекритим). Після цієї перевірки ми дивимся на збережене нами значення змінної «rerstart», якщо ребро було перекрите, то ми знову

викликаємо функцію найкорошого шляху з модуля Дейкстри і переходимо на наступну ітерацію циклу. Якщо ж ребро було відрите, ми по ньому ідемо. Код, який реалізує даний алгоритм наведено у **додатку Д.**

РОЗДІЛ 5 ПОРІВНЯННЯ ЕФЕКТИВНОСТІ

В цьому розділі наведено дані отримані за допомогою програми, описаної в попередньому розділі. Графіки та таблиці побудовані в застосунку Excel. Також цей розділ поділив на окремі підрозділи для того, щоб подивитись як окремо кожен з параметрів впливає на роботу алгоритмів. Оскільки, при генерацій графу ми додаємо першу та останню вершину, то це і будуть початкова та кінцева точки відповідно.

5.1 ЗАГАЛЬНЕ ПОРІВНЯННЯ

Для порівняння я згенерував граф з кількістю вершин 30 та відсотком закритих доріг 20% (вони позначені червоним кольором) та кількістю ітерацій 10 – це означає, що коли згенерується граф і всі алгоритми завершать свою роботу, тоді програма заново випадковим чином вибере інші ребра для блокування і так десять разів. Ми отримали такий граф на першій ітерації - рис. 24, та такі дані табл. 1. Більшість алгоритмів обрали такий шлях: 0-14-5-19-15-6-29. Average cost – середнє значення по загальній вартості шляхів, які обирав алгоритм за всі ітерації. Maximum cost – максимальне значення загальної вартості шляху за всі ітерації, а Minimum cost – відповідно мінімальне. Average time – середнє значення часу, який витрачав алгоритм на пошук шляху, вимірюється в секундах.

Strategy	Average Cost	Maximum Cost	Minimum Cost	Average Time
Greedy	50.91457913	58.32287953	47.50662466	0.01479706
Alternative Greedy	50.91457913	58.32287953	47.50662466	0.0189431
Waiting	55.00662466	56.50662466	47.50662466	0.00680154
Alternative Waiting	50.05127526	53.85074357	47.50662466	0.0134978

таблиця 1

З таблиці видно, що в цьому випадку, жадібний алгоритм в середньому працює однаково з своєї альтернативною версію, як за часом так і за вартістю пройденого шляху. Варто зазначити, що показник minimum cost однаковий у

всіх алгоритмів, адже при великій кількості ітерацій в якийсь момент ребра у графі будуть перекриті так, що не будуть належати найкоротшому шляху в графі, тому в усіх алгоритмах не буде випадку, коли їм треба змінювати шлях і вони будуть ідти по шляху, який їм передав алгоритм Дейкстри, коли ця функція викликалась вперше. Також з табиці видно, що альтернативний алгоритм «очікуваня» працює трохи краще ніж просто алгоритм очікування, якщо порівнювати показник середньої вартості шляху, проте для його роботи знадобилось більше часу. Це може бути викликано тим, що ми набагто частіше викликали функцію пошуку найкоротшого шляху з модуля Дейкстри ніж в звичайному алгоритмі «очікування», де ми це робимо всього один раз. Серед усіх алгоритмів можна назвати лідером лише альтернативний алгоритм «очікування», бо в нього найкраші показники, як за часом так і за середньою вартістью шляху.

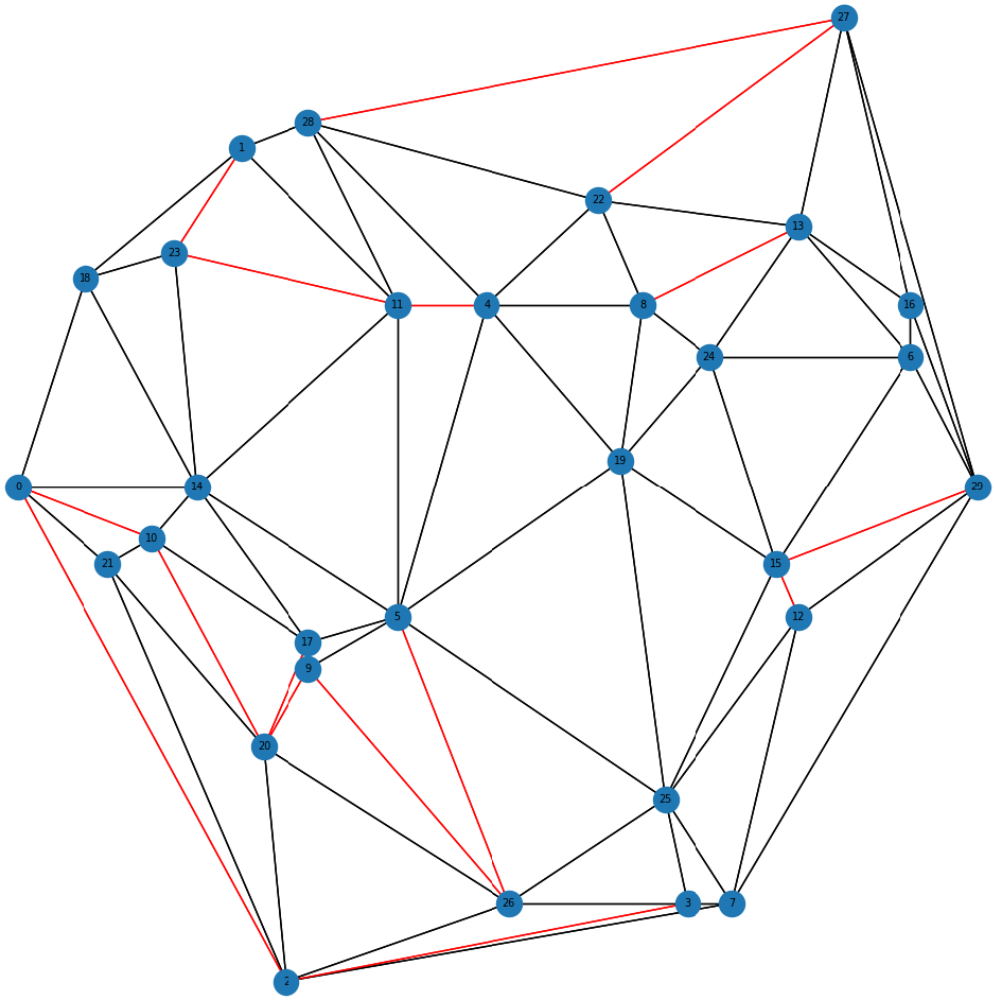
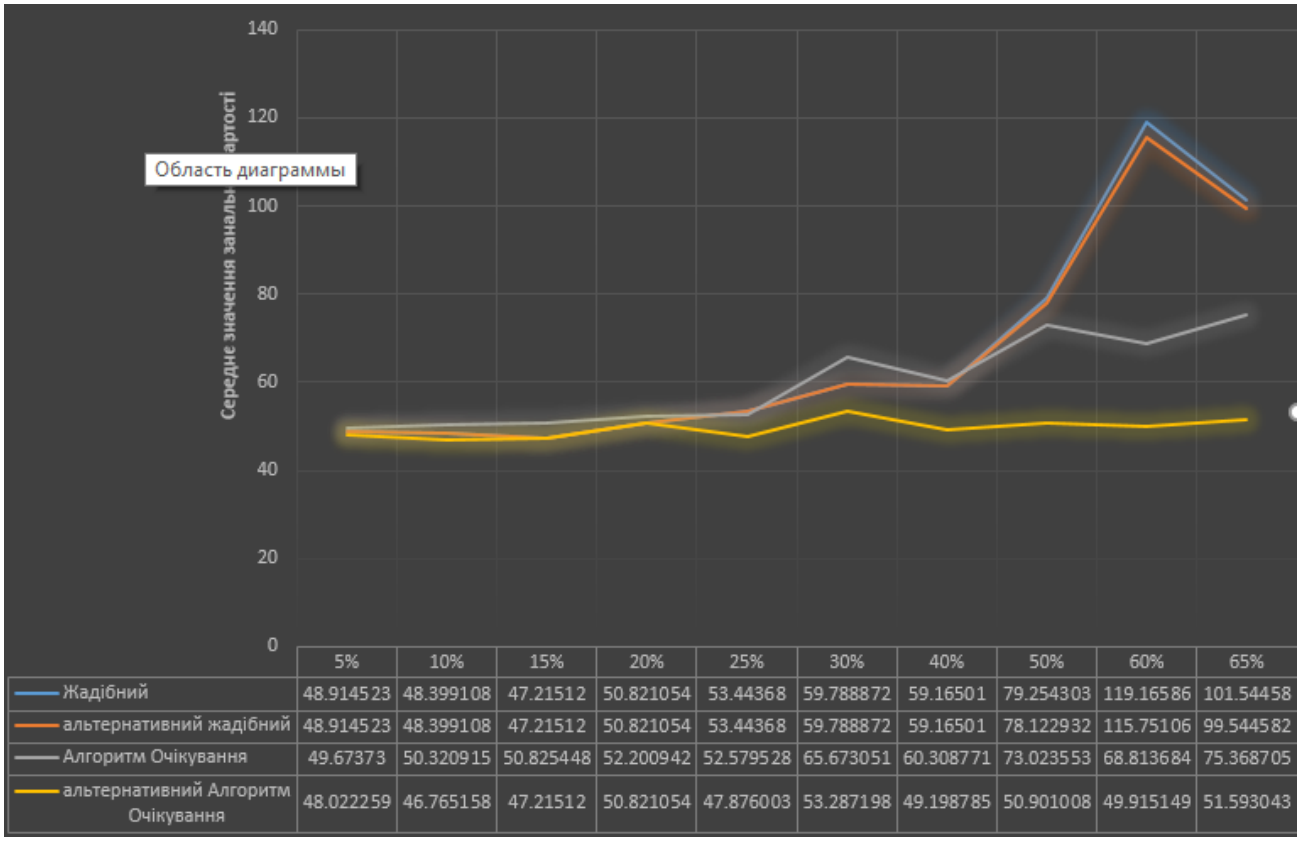


рисунок 21

5.2 ПОРІВНЯННЯ ЗА ВІДСОТКОМ ЗАКРИТИХ ДОРІГ

В данному розділі я вирішив подивитись, як себе покажуть алгоритми, якщо ми почнемо збільшувати відсоток закритих доріг, проте не до дуже високих значень, бо зростає ймовірність того, що при генерації закриття доріг утвориться розрив графу і ми не зможемо проходити у кінцеву точку, а так данні не будуть інформативними. Оскільки при зміні відсотка закритих доріг программа заново виконує генерацію графа, тому я вирішив залишити кількість вершин графа незмінною, таким чином при новій генерації у нас точки змінюватиму положення на площині, проте це не впливає на кількість ребер, а лише на їх розміщення на площині. Для перевірки було взято граф з тридцятьма

вершинами, десятьма ітераціями, що дозволить більш спраедливо оцінювати роботу. Дані відображенно на рис. 25.



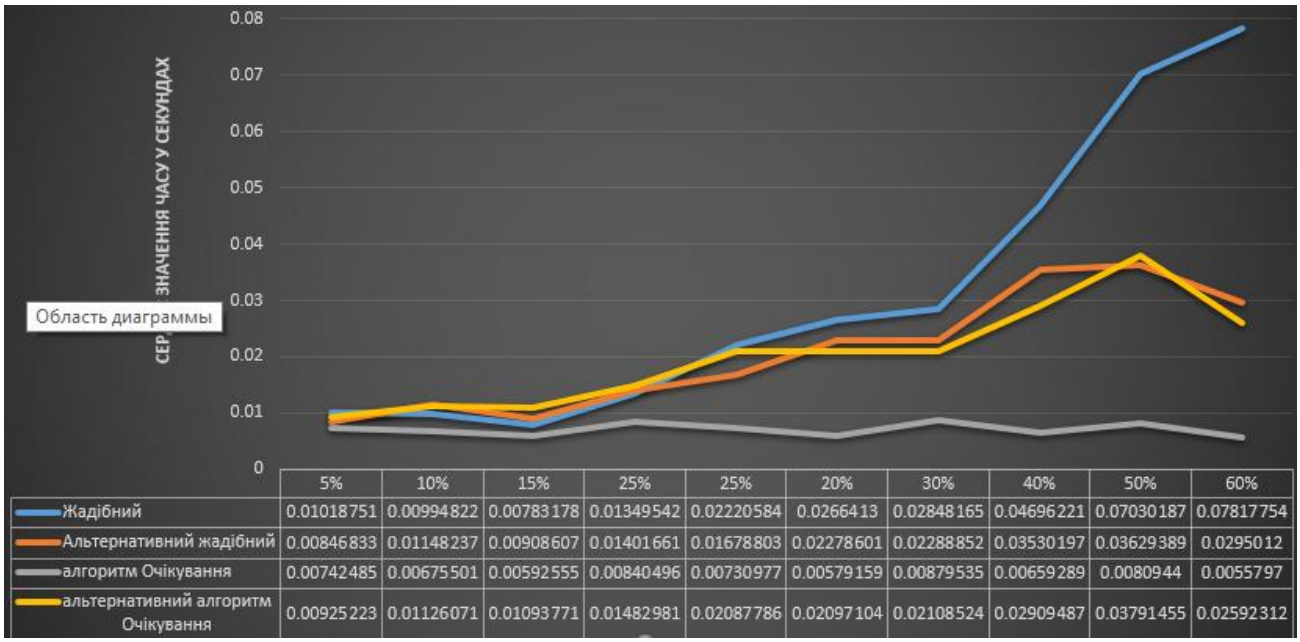
	5%	10%	15%	20%	25%	30%	40%	50%	60%	65%
Жадібний	48.914523	48.399108	47.21512	50.821054	53.44368	59.788872	59.16501	79.254303	119.16586	101.54458
альтернативний жадібний	48.914523	48.399108	47.21512	50.821054	53.44368	59.788872	59.16501	78.122932	115.75106	99.544582
Алгоритм Очікування	49.67373	50.320915	50.825448	52.200942	52.579528	65.673051	60.308771	73.023553	68.813684	75.368705
альтернативний Алгоритм Очікування	48.022259	46.765158	47.21512	50.821054	47.876003	53.287198	49.198785	50.901008	49.915149	51.593043

рисунок 22

Як ми бачимо з графіку при невеликому відсотку перекриття доріг, алгоритми показують доволі однаковий результат за середнєм значенням загальної вартості шляху. Проте після позначки в п'ятдесят відсотків уже бачимо зміни, а саме: альтернативний жадібний алгоритм знаходить шляхи більш дешевші, а ніж жадібний, а альтернативний алгоритм «очікування» усіх випадках показує приблизно ідеальні результати і значно краще ніж звичайний алгоритм «очікування».

На рис. 26 відображено часові показники, відносно росту відсотку заблокованих доріг для кожного алгоритму. Якщо порівнювати обидва жадібних алгоритма за часом, то альтернативний має кращі результати, хоча на кожній ітерації ми перевіряємо усі інцидентні ребра, що мало

вплинути на час, проте як ми бачимо - це не значно влпиває на час та ми маємо більше інформації, що економить нам час для пошуку уже нового шляху, коли ми зустрічаємо на шляху перекриту дорогу. Як і очікувалось алгоритм «очікування» дав досить гарні показники по часу, адже він досить простий і ми не беремо до уваги перекриті ребра, а альтернативний має більші часові витрати, бо ми робимо перевірки на кожній ітерації та вразі перекриття дороги виклиаємо алгоритм Дейкстри для пошуку нового шляху.

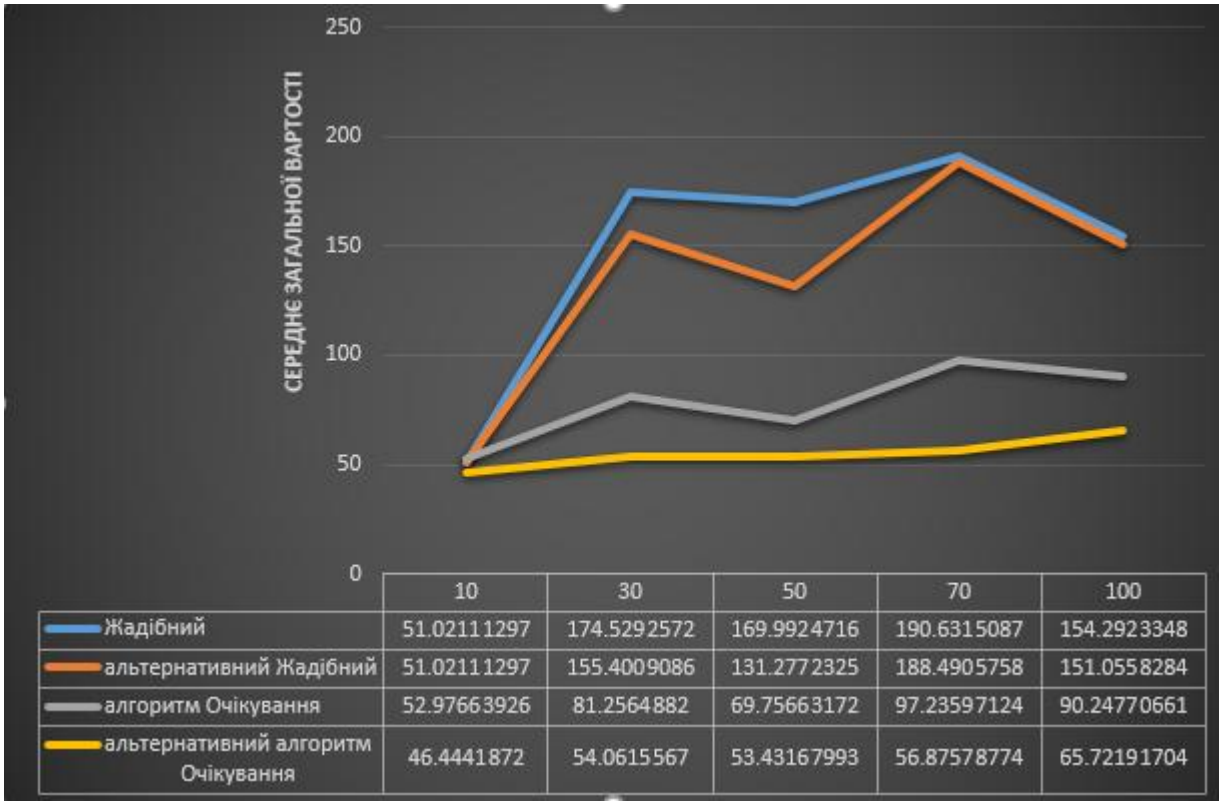


	5%	10%	15%	25%	25%	20%	30%	40%	50%	60%
Жадібний	0.01018751	0.00994822	0.00783178	0.01349542	0.02220584	0.0266413	0.02848165	0.04696221	0.07030187	0.07817754
Альтернативний жадібний	0.00846833	0.01148237	0.00908607	0.01401661	0.01678803	0.02278601	0.02288852	0.03530197	0.03629389	0.0295012
алгоритм Очікування	0.00742485	0.00675501	0.00592555	0.00840496	0.00730977	0.00579159	0.00879535	0.00659289	0.0080944	0.0055797
альтернативний алгоритм Очікування	0.00925223	0.01126071	0.01093771	0.01482981	0.02087786	0.02097104	0.02108524	0.02909487	0.03791455	0.02592312

рисунок 23

5.3 ПОРІВНЯННЯ ЗА КІЛЬКОСТЮ ВЕРШИН В ГРАФІВ

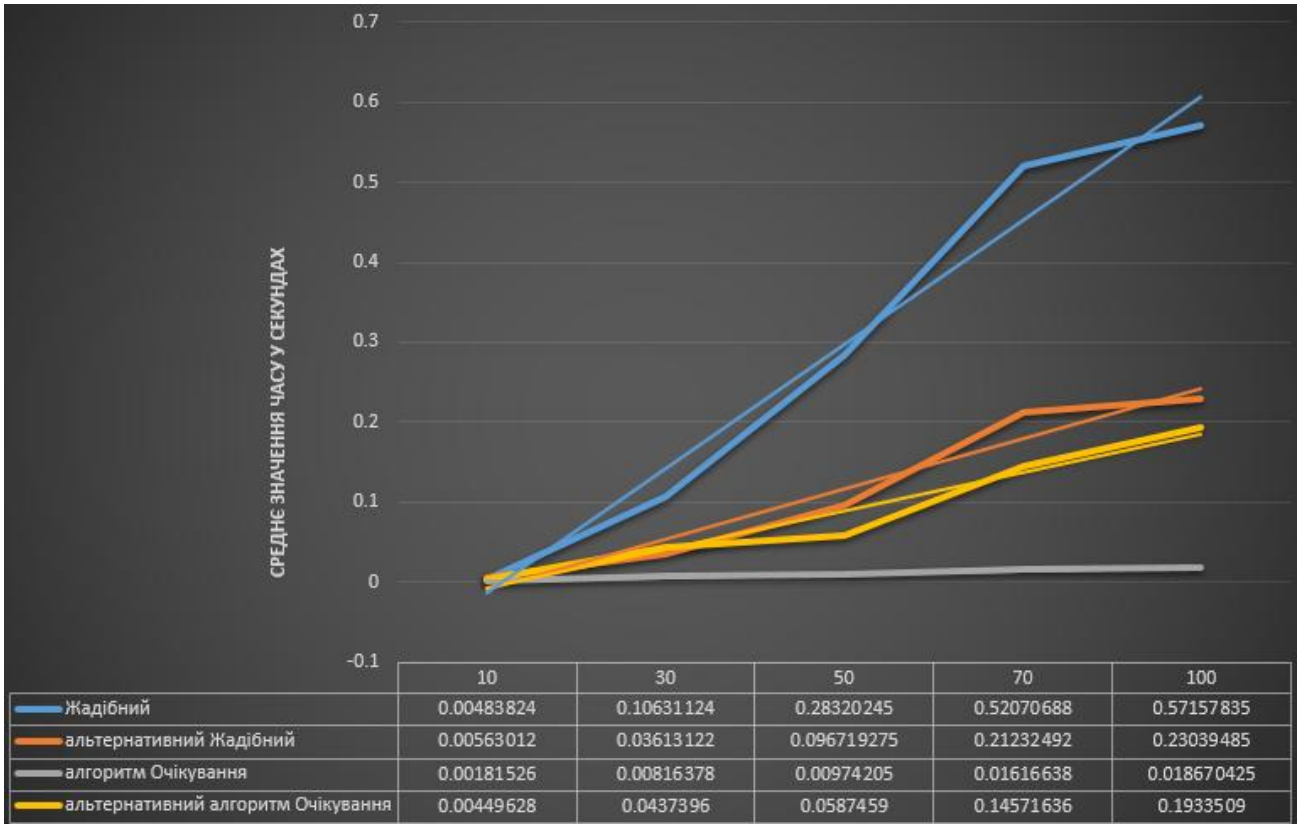
У цьому підрозділі я вирішив подивитись, як будуть працювати алгоритми, якщо ми зафіксуємо кількість блокованих доріг та лінійно будем збільшувати кількість вершин, що належать графу. Таким чином можна дослідити, як впливає кількість вершин на вибір маршруту та як змінюється час виконання пошуку цього шляху. Відсоток доріг я вирішив виставити 65%, адже на такому показнику ми побачили зміни та різні значення в роботі алгоритмів, тому було обрано саме таке число та кількіть ітерацій 5.



	10	30	50	70	100
Жадібний	51.02111297	174.5292572	169.9924716	190.6315087	154.2923348
альтернативний Жадібний	51.02111297	155.4009086	131.2772325	188.4905758	151.0558284
алгоритм Очікування	52.97663926	81.2564882	69.75663172	97.23597124	90.24770661
альтернативний алгоритм Очікування	46.4441872	54.0615567	53.43167993	56.87578774	65.72191704

рисунок 24

Тож на рис. 27 видно, що в цілому вибір шляху також зростає лінійно для обох алгоритмів «очікування», як і кількість вершин графу. Також можна побачити скачок, який видно для обох жадібних алгоритмів на позначці 30, що показує, як функції зростають, проте їх ріст не можна назвати лінійним.



	10	30	50	70	100
Жадібний	0.00483824	0.10631124	0.28320245	0.52070688	0.57157835
альтернативний Жадібний	0.00563012	0.03613122	0.096719275	0.21232492	0.23039485
алгоритм Очікування	0.00181526	0.00816378	0.00974205	0.01616638	0.018670425
альтернативний алгоритм Очікування	0.00449628	0.0437396	0.0587459	0.14571636	0.1933509

рисунок 25

На рис. 28 зображено графіки, що показують, як змінюється середнє значення часу від кількості вершин. Також на графіки додані лінії тренду, які є лінійними, що спрощує роботу, та наглядно показує для кожного алгоритму, що з лінійним ростом кількості вершин, витрачений час на пошук шляху також росте лінійно.

5.4 АНАЛІЗ ОТРИМАНИХ ДАНИХ

З отриманих даних ми можемо побачити, що з лінійною ростом значень таких параметрів, як кількість вершин у графі та кількість заблокованих доріг, час виконання програми росте лінійно, чітко видно переваги альтернатиних алгоритмів жадібного та алгоритму «очікування», що показують кращі результати, як за часом так і за середнім показником вартості шляху. Це в свою чергу показує, що зібрана додаткова інформація під час перевірки інцидентних ребер впливає на ефективність виконання. Також під час виконання роботи видно, що альтернативний алгоритм «очікування» проходить шлях, який близький до оптимального, що вказує на високу ефективність цього алгоритму. Також варто відмітити, що порівнювати треба лише окремо жадібний та альтернативний жадібний, і алгоритм «очікування» та альтернативний алгоритм «очікування», адже вони вирішують кардинально різні задачі. Тому для вибору вирішення стохастиної версії задачі канадського водія варто використовувати альтернативний жадібний алгоритм, а для вирішення цієї задачі, коли ребра не є закритими, а мають «штраф» варто використовувати альтернативний алгоритм «очікування».

Варто відмітити, що показники часу мають велику залежність від якості написання коду, тому їх можна поркащити, вдосконаливши код програми, або використовуючи компільовані мови програмування. Також я використовував практично тіки одну стратегію, проте можливо в якості основного алгоритму можна було обрати інший (наприклад A*), що могло вплинути на роботу програми в цілому, оскільки основний алгоритм моєї програми – це алгоритм Дейкстри.

ВИСНОВКИ

Мною було опрацьовано теоретичний матеріал з тем: «задача канадського водія», «пошук найкоротшого шляху в графі». Було реалізовано програму і окремі модулі, що реалізують роботу алгоритмів. Після написання програми було виконанно аналіз отриманих даних, аналіз коду програми. Чисельні дані підтверджують інформацію про ефективність альтернативних алгоритмів для вирішення даної постановки задачі. Дані методи можуть бути вдосконалені шляхом використання компілльованої мови програмування та більш кращих бібліотек для роботи з такою структурою даних, як граф. Також корисним може бути вдосконалення алгоритму Дейкстри, щоб він міг «запам'ятовувати» не змінені мінімальні шляхи, що дасть значний виграш в часі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Papadimitriou C.H., Yannakakis M. Shortest paths without a map // Theoretical Computer Science. – 1991. – v. 84.– pp. 127-150.
2. Lim Zh., Hsu D., Lee S. Shortest Path under Uncertainty: Exploration versus Exploitation. // Conference on Uncertainty in Artificial Intelligence. Proceedings. – 2017. – Режим доступу до ресурсу: https://bigbird.comp.nus.edu.sg/m2ap/wordpress/wp-content/uploads/2017/08/uai17.pdf
3. Dror F., Shimony S., Benbassat A., Wenner C. Complexity of Canadian traveler problem variants // Theoretical Computer Science. – 2013. – v. 487. – pp. 1-16. – Режим доступу до ресурсу: https://www.sciencedirect.com/science/article/pii/S0304397513002296]
4. Canadian Traveller Problem - Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Canadian_traveller_problem
5. Мова програмування python. Офіційна документація версії 3.7 [Електронний ресурс] – Режим доступу до ресурсу: https://docs.python.org/3.7/
6. Жадібний алгоритм. [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Greedy_algorithm
7. Алгоритм Дейкстри. Пошук найкоротшого шляху [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/111361/
8. Guo H., Barfoot Th. The Robust Canadian Traveler Problem Applied to Robot Routing // 2019 International Conference on Robotics and Automation (ICRA). – 2019. Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/8794252/authors#authors

ДОДАТКИ

ДОДАТОК А

```
    def create(self, n):
        #створюємо список вершин
        nodes = []

        for i in range(n):
            nodes.append(i)
        points = []
        # виставляємо першу вершину
        points.append((0, 22))
        # випадковим чином генеруємо координати
        for j in range(1, len(nodes) - 1):
            x = random.randint(3, 40)
            y = random.randint(3, 40)
            while ((x, y)) in points:
                x = random.randint(3, 40)
                y = random.randint(3, 40)
            points.append((x, y))
        # додаємо останню вершину
        points.append((43, 22))
        #тріангуляція Делоне
        t = Delaunay(points)
        edges = []
        m = dict(enumerate(nodes))
        for i in range(t.nsimplex):
            edges.append((m[t.vertices[i, 0]], m[t.vertices[i, 1]]))
            edges.append((m[t.vertices[i, 1]], m[t.vertices[i, 2]]))
            edges.append((m[t.vertices[i, 2]], m[t.vertices[i, 0]]))
        #створюємо об'єкт, тобто сам граф та будуємо ребра.
        G = nx.Graph()
        for i in range(len(edges)):
            G.add_edge(edges[i][0], edges[i][1], color="black")
        #словник для кожної вершини
        pos = dict(zip(nodes, points))
        nx.set_node_attributes(G, pos, "pos")
        #виставляємо властивості для ребер
        for i in range(len(edges)):
            c1 = pos[ed1[i]]
            c2 = pos[ed2[i]]
            G[ed1[i]][ed2[i]]["weight"] = math.sqrt(((c2[0] - c1[0]) ** 2) + ((c2[1] -
c1[1]) ** 2))
            G[ed1[i]][ed2[i]]["penalty"] = random.randint(5, 10)
            G[ed1[i]][ed2[i]]["propability"] = random.uniform(0, 0.6)

        edges = G.edges()

        colors = []
        for i in range(len(edges)):
            colors.append("black")

        self.graph = G
               return G
```

ДОДАТОК Б

```
def generate_blockades(self, block_percent):
    # block_percent - загальний відсоток заблокованих доріг
    G = self.graph.copy()

    # так позначимо всі відкриті дороги в графі, замалювавши їх чорним
    nx.set_edge_attributes(G, "black", "color")

    # рахуєм кількість доріг, яку необхідно закрити
    count = round(len(G.edges()) / 100 * block_percent)

    # випадковим чином вибираємо ребра зі списку усіх
    blocked = random.sample(list(G.edges()), count)

    # замальовуємо їх червоним кольором
    for e in blocked:
        G[e[0]][e[1]]["color"] = "red"
           return G
```

ДОДАТОК В

```
def get_path(self, source, target):
    path = []
    path.append(target)
    while target != source:
        target = self.graph.nodes[target]["previous"]
        path.append(target)

    path.reverse()

    return path

def get_path_length(self, path):
    try:

        sz = len(path)
        cost = 0

        for i in range(0, sz - 2):
            cost += self.graph.get_edge_data(path[i], path[i + 1])["weight"]

        cost += self.graph.get_edge_data(path[sz - 2], path[sz - 1])["weight"]

    except TypeError:
        pass

    return cost

def shortest_path(self, source, target):
    nx.set_node_attributes(self.graph, None, "previous")
    nx.set_node_attributes(self.graph, sys.maxsize, "dist")
    nx.set_node_attributes(self.graph, False, "visited")

    visited = []
    queue = []

    self.graph.nodes[source]["dist"] = 0

    heapq.heappush(queue, (0, source))

    while len(queue) != False:
        current = heapq.heappop(queue)
        self.graph.nodes[current[1]]["visited"] = True

        visited.append(current[1])

        for neighbor in self.graph.neighbors(current[1]):

            if self.graph.nodes[neighbor]["visited"] == False:
                distfromstart = (
                    self.graph.nodes[current[1]]["dist"]
                    + self.graph.get_edge_data(current[1], neighbor)["weight"])

            if distfromstart < self.graph.nodes[neighbor]["dist"]:
                self.graph.nodes[neighbor]["dist"] = distfromstart
                self.graph.nodes[neighbor]["previous"] = current[1]
                heapq.heappush(queue, (distfromstart, neighbor))

           return self.get_path(source, target)
```

ДОДАТОК Г

```
def search(self, source, target):
        G = self.graph.copy()

        start = timer()

        path = self.shortest_path(source, target)

        route = []
        route.append(path[0])

        i = 0

        while route[-1] != target:
            if self.graph.get_edge_data(path[i], path[i + 1])["color"] != "red":
                route.append(path[i + 1])
                i += 1
            else:
                self.graph.remove_edge(path[i], path[i + 1])
                path = self.shortest_path(path[i], target)
                i = 0


        end = timer()
        elapsed = end - start

        print("Search time:", elapsed)
        print("Path: ", route)
        print("Path length:", self.get_path_length(route))

        self.graph = G

               return elapsed, self.get_path_length(route), route
```

ДОДАТОК Д

```
def search(self, source, target):
    G = self.graph.copy()

    start = timer()

    path = self.shortest_path(source, target)

    route = []
    route.append(path[0])

    penalty = 0

    i = 0

    while route[-1] != target:
        if self.graph.get_edge_data(path[i], path[i + 1])["color"] != "red":
            pass
        else:
            penalty += self.graph.get_edge_data(path[i], path[i + 1])["penalty"]

        route.append(path[i + 1])
        i += 1

    end = timer()
    elapsed = end - start

    print("Search time:", elapsed)
    print("Path: ", route)
    print("Path length:", self.get_path_length(route) + penalty)

    self.graph = G

    return elapsed, (self.get_path_length(route) + penalty), route
```