**Київський національний університет**

**імені Тараса Шевченка**

Факультет комп'ютерних наук та кібернетики

Кафедра теорії та технології програмування

**Звіт до лабораторної роботи**

за спеціальністю 122 Комп’ютерні науки “Інформатика”

на тему:

**«Найкоротші шляхи на множині перешкод у 2D»**

Виконав студент 3-го курсу групи ТТП-32

Остренко Олександр Максимович

АНОТАЦІЯ

За результатами виконаної роботи, автор реалізував власний алгоритм для пошуку найкоротшого шляху між двома точками. Алгоритм є чуттєвим до щільності полігонів і в окремих випадків може досягати експоненціальної складності. На думку автора алгоритм є достатньо перспективним, але потребує подальшого доопрацювання.

Based on the results of the work performed, the author implemented his own algorithm for finding the shortest path between two points. The algorithm is sensitive to the density of polygons and in some cases can reach exponential complexity. In the opinion of the author, the algorithm is quite promising, but needs further refinement.

ЗМІСТ

[АНОТАЦІЯ 2](#_Toc167290151)

[ЗМІСТ 3](#_Toc167290152)

[ВСТУП 4](#_Toc167290153)

[ОСНОВНА ЧАСТИНА 6](#_Toc167290154)

[Складність запропонованого алгоритму. 10](#_Toc167290155)

[ПРАКТИЧНА ЧАСТИНА 11](#_Toc167290156)

[Особливості програмної реалізації. 11](#_Toc167290157)

[Опис основних функцій. 11](#_Toc167290158)

[Ввід-вивід даних. 12](#_Toc167290159)

[Лістінг основних модулів програми. 13](#_Toc167290160)

[ВИСНОВКИ 15](#_Toc167290161)

[ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 17](#_Toc167290162)

[ДОДАТКИ 18](#_Toc167290163)

[ДОДАТОК А. 18](#_Toc167290164)

[ДОДАТОК Б. 18](#_Toc167290165)

[ДОДАТОК В. 18](#_Toc167290166)

[ДОДАТОК Г. 19](#_Toc167290167)

[ДОДАТОК Д. 19](#_Toc167290168)

[ДОДАТОК Е 20](#_Toc167290169)

[ДОДАТОК Є. 20](#_Toc167290170)

[ДОДАТОК Ж. 21](#_Toc167290171)

ВСТУП

Пошук найкоротшого шляху на множині перешкод достатньо цікава задача, вона може використовуватись в комп’ютерних іграх, коли треба розрахувати найкоротшу траєкторію від гравця до скарбу, або ж коли треба, щоб ігровий персонаж якомога швидше догнав гравця і тд.

Найчастіше пропонують представити досліджувану область у виді графу, а далі використовувати алгоритми пошуку найкоротшого шляху на графі, такі як алгоритм Дейкстри[1], або ж А\*[2] алгоритм. При даних підходах залишається питання – яким чином максимально ефективно представити досліджувану область у вигляді графу? Була ідея зробити це за допомогою тріангуляції, а далі використати той же алгоритм Дейкстри[1] для пошуку мінімального шляху, але тріангуляція вона не враховує наше бажання побудувати мінімальний шлях, і нема ніякої гарантії, що кінцевий шлях буде оптимальним.

Також автор розглядав алгоритм хвильової трасировки   
(Алгоритм Лі)[3], але даний алгоритм теж не дуже підходить під умови задачі, бо він розбиває область дослідження на квадратики, і шукає найкоротший шлях переміщаючись по даним квадратикам.

Изображение выглядит как кроссворд, прямоугольный, текст, снимок экрана

Автоматически созданное описание

Тут виникає два питання – як масштабувати розміри квадратиків і чи можна вважати отриманий шлях мінімальним, якщо, в загальному, мінімальний шлях між двома точками – пряма, а при даному підході у нас множина квадратиків? Оригінально, даний алгоритм використовувався для оптимального з’єднання розташованих на платі елементів, відповідно друге питання не було актуальним, а перше вирішувалось по ходу роботи, але для нашої задачі алгоритм не є вдалим.

Після того, як автор не зміг знайти підходящий алгоритм, було прийнято рішення розробити свій алгоритм. Глобально, ідея в тому, щоб знайти перший полігон, вибрати його крайові (відносно прямої АВ) точки, і спробувати «обійти» даний полігон з двох сторін.

Изображение выглядит как линия, диаграмма, скат

Автоматически созданное описание

Алгоритм виконується рекурсивно, і в кінці-кінців отримуємо список різних шляхів та відповідних довжин, після чого дістаємо найкоротший.

**ОСНОВНА ЧАСТИНА**

Задача – знайти найкоротший шлях між точками, який би не перетинав жодного полігону на площині.

Аксіома 1: Найкоротша відстань між двома точками – пряма.

Аксіома 2: Для довільного трикутника ABC завжди буде виконуватись рівність – AB + BC>= AC.

Відповідно до цих двох аксіом, найкоротшим обходом полігона буде саме ламана в самій дальній точці полігону, а ніякі не полукола чи «дійшли, по краю обійшли та пішли далі».

Алгоритм складається з двох ключових етапів – побудувати усі можливі шляхи з урахуванням обходу полігонів з обох сторін, після чого визначити найкоротший шлях.

Пошук мінімального шляху дуже простий – пройтись по всім шляхам списку і визначити мінімальний. Складність даного алгоритму О(n).

Для побудови усіх можливих шляхів, будемо використовувати наступний алгоритм:

1. На самому початку проводимо пряму AB через дві точки.
2. Перебираємо усі полігони і перевіряємо, чи перетинає пряма AB даний полігон – чи є в прямій AB та лініями многокутника спільні точки? Якщо так, то перевіряє, чи дана пряма є дотичною для даного многокутника. Якщо є спільна точка, але пряма не є дотичною, значить пряма перетинає даний полігон – додаємо його до списку.
3. Проходимо по кожному полігону, який перетинає пряма AB та знаходимо відстані від точки A до перших сегментів, що перетинаються прямою, кожного з полігонів. Після чого вибираємо найближчий до точки A.
4. Для найближчого полігону треба визначити крайові точки відносно прямої AB. Щоб не перебирати всі його вершини і не рахувати відстань від кожної вершини до прямої (підрахунок відстаней доволі довга операція, бо використовує квадратний корінь), автором було запропоновано наступний алгоритм:
   1. Визначити найближчу до точки А сторону многокутника, яку перетинає пряма AB. Щоб визначити таку, слідуємо правилу – якщо нормаль відповідного сегменту (нормалі визначаємо проти годинникової стрілки, відповідно вона направлена назовні многокутника) співспрямована з вектором прямої AB (співспрямована => скалярний добуток >= 0), то даний сегмент дальній, інакше – ближній.
   2. Далі переходимо по сусіднім сегментам за та проти годинникової стрілки та перевіряємо кут між вектором прямої та нормаллю сегмента – якщо відповідний кут <= 90 градусів => скалярне множення >= 0 => точка крайова.
5. Після визначення крайових точок, дублюємо пряму AB на дві ломані – ACB та ADB, де C та D відповідні крайові точки.
6. Треба перевірити, чи прямі AC та AD не перетинають жодний полігон. Якщо ні – рекурсивно пророблюємо даний алгоритм для прямих CB та DB. Якщо перетинають, то рекурсивно пророблюємо даний алгоритм для прямих AC чи/та AD.
   1. Окремий випадок – коли полігон «незручної» форми, тобто коли його не можливо обійти одним «заломом».

Изображение выглядит как линия, диаграмма, График, скат

Автоматически созданное описание

В такому випадку нам його треба огинати. Щоб робити це максимально ефективно, будемо запам’ятовувати, з якої сторони ми починаємо обхід полігону. Тобто введемо деякий список пар («полігон», «0-зліва, 1-справа»). Так як алгоритм рекурсивний, то в кожному альтернативному просторі ми будемо огинати полігон тільки з одної сторони

1. Після кожного успішного обходу спрацьовує крок оптимізації. Він потрібен для випадку, коли ломану можна згладити:

Изображение выглядит как линия, диаграмма, График, скат

Автоматически созданное описание

(1)

Изображение выглядит как линия, диаграмма, скат, График

Автоматически созданное описание

(2)

Після реалізації описаного вище алгоритму, автором було підмічено один значний недолік – якщо щільність полігонів була достатньо велика, то алгоритмом будувались шляхи, дуже далекі від початкової прямої. Тоді було прийнято рішення обмежити область пошуку за наступним принципом.

Изображение выглядит как диаграмма

Автоматически созданное описание

На малюнку позначено точки А та В, між якими треб прокласти найкоротший шлях. Для початку, серед усіх полігонів, що перетинає пряма АВ, треба знайти найвіддаленіші відносно АВ дві точки (по обидві сторони від прямої). Після того будуємо допустиму область пошуку – подвійні найбільші відстані (між синіми лініями). Відповідно, візьмемо за аксіому гіпотезу – розв’язок задачі буде завжди знаходитись в даному тунелі.

Чому саме подвійні відстані? На малюнку видно, щоб обійти полігон 1 зліва, нам треба спочатку обійти полігон 2. Так як ліва крайня точка полігону 2 виходить за межі тунелю, то ми розуміємо, що не доцільно досліджувати ту область, бо економніше буде обійти справа – вернутись ближче до прямої, ніж робити обхід зліва. Але якщо така відстань < 2\*h, то треба такий випадок теж дослідити.

## Складність запропонованого алгоритму.

Даний алгоритм є дуже вразливим до щільності полігонів. На розрідженому просторі пошук виконується достатньо швидко, але в іншому випадку алгоритм зависає, бо треба побудувати бо два шляхи обходу для кожного полігону. В найгіршому випадку, коли обходити треба буде абсолютно кожен полігон, складність буде прямувати до O(2^h), де h – кількість полігонів. В загальному випадку складність буде меншою, але результат все одно є достатньо сумним.

ПРАКТИЧНА ЧАСТИНА

## Особливості програмної реалізації.

Для опрацювання геометричних фігур, було прийнято рішення використати бібліотеку shapely.geometry. Вона зручна тим, що представляє опис таких структур як точка, лінія, полігон, і реалізує деякі функції – пошук відстаней, перетинів і тд.

## Опис основних функцій.

Перша підзадача – псевдовипадковим алгоритмом створити потрібну кількість полігонів. Для вирішення даної підзадачі автор використовує бібліотеку random, за допомогою якої створює набір точок потрібної розмірності. [ДОДАТОК А]

Для побудови потрібної кількості полігонів, автор використовує бібліотеку sklearn.cluster для кластеризації згенерованих точок відповідну кількість кластерів, після чого використовує бібліотеку scipy.spatial для побудови опуклої оболонки до кожного кластера. Таким чином, створимо потрібну кількість полігонів. [ДОДАТОК Б]

Проблема – якщо співвідношення кількості точок до кількості полігонів >= 4, то з великою ймовірністю значна кількість точок буде «поглинута» даним алгоритмом, бо буде всередині опуклої оболонки. Рекомендація – щоб відповідне співвідношення було 3 або 4, тоді буде виведено і потрібну кількість полігонів, і потрібну кількість точок.

Основна проблема – прокласти найкоротший шлях між точками. Для визначення всіх полігонів, що перетинає лінія, автор вирішив використати можливості бібліотеки shapely.geometry [ДОСДАТОК В]. Для пошуку найближчого полігона з тих, що перетинає лінія, автор теж використав можливості вище зазначеної бібліотеки [ДОДАТОК Г].

Для пошуку крайових точок було вирішено реалізувати наведений в основній частині алгоритм. В [ДОДАТОК Д] наведено реалізацію обходу полігона та визначення одної крайової точки – визначаємо нормаль сегменту (пари точок) та перевіряємо скалярний добуток на умову >= 0, щоб зрозуміти, чи даний сегмент є крайовим. Частину реалізації основної функції, що огинає полігони, наведено в [ДОДАТОК Е]. Там продемонстровано як проходить рекурсивний обхід полігона з врахуванням того, з якої сторони ми розпочали його обхід і чи розпочали взагалі (стр. 167-171). В [ДОДАТОК Є] показано як саме оптимізується успішно пророблений обхід – рекурсивно проходимо по всім побудованим під час обходу точкам і видаляємо неактульні.

В [ДОДАТОК Ж] наведена реалізація тунелю – послідовно проходимо по всім крайовим точкам і знаходимо дві найдальші.

## Ввід-вивід даних.

Передбачено два можливих випадки – ввід опис полігонів через .txt файлів, та за допомогою автоматизованого алгоритму. Файл .txt має бути заповнений за наступним шаблоном.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Спочатку пишеться ключове слово, після чого, парами, записуються координати x, y відповідних точок. Важливо – перша та остання точка має збігатись, щоб позначити замкнутий полігон.

Точки, між якими будується найкоротший шлях, задаються динамічно – під час виконання програми треба клацнути лівою кнопкою миші полі Plot.

Вивід даних іде в два потоки – графічний вивід можна побачити на Plot, статистику (час, використаний на побудову полігонів, на пошук можливих шляхів та на визначення найкоротшого шляху) можна побачити в консолі.

Изображение выглядит как текст, диаграмма, линия, График

Автоматически созданное описание

Изображение выглядит как текст, Шрифт, снимок экрана, типография

Автоматически созданное описание

На графічному інтерфейсі Plot зображено синю пряму від точки до точки, та чорну ломану – найкоротший шлях.

## Лістінг основних модулів програми.

1. Main.py – файл, в якому зосереджена вся комунікація з користувачем та розміщений графічний двіжок.
2. Helper.py – файл, в якому зосереджені допоміжні класи.
   1. Random – клас, відповідальний за генерацію випадкових точок та подальшу побудову полігонів, на цій множині точок.
   2. File – клас, відповідальний за зчитування/запис полігонів з/до файлу.
   3. Painter – клас, що використовувався при налагодженні програми, - малює статичний Plot з проміжними результатами.
3. MagicBall.py – файл, в якому зосереджені класи для реалізації самого алгоритму.
   1. Helper – клас, реалізуючий векторні операції – знаходження вектора по двом точкам, знаходження нормалі до вектора, та знаходження скалярного добутку векторів.
   2. MagicBall – клас, в якому зосереджені функції, що разом реалізують алгоритм.

ВИСНОВКИ

В загальному, не можна сказати, що запропонований алгоритм є універсальним. Складність алгоритму дуже сильно залежить від щільності полігонів, бо чим більше щільність, тим більше рекурсивних обходів треба зробити, що є значним недоліком. Алгоритм достатньо паршиво проявляє себе коли кількість полігонів досягає 1000 на квадратному полі 1000\*1000.

Изображение выглядит как текст, снимок экрана, Прямоугольник

Автоматически созданное описание

Даний, достатньо простий пошук, виконувався цілих 30 секунд. Але, при невеликій щільності, алгоритм показує себе достатньо хорошо.

Для початку, автор запустив програму-профіліровщик і помітив, що найбільше часу витрачається на фрагмент коду в [ДОДАТОК В]. Відповідно перше, що бажано зробити, - це відмовитись від бібліотеки та написати свої, більш ефективні методи, для пошуку полігонів, що перетинає пряма.

Далі, можна було б подумати над тим, як на корені відсікати випадки, коли ломана будується як змійка (з дальнього лівого краю на дальній правий і тд.), бо на всі такі випадки витрачається час, хоча вони апріорно є хибними. Також можна було б подумати над тим, як можна апріорно оцінити найкращу сторону для обходу полігона.

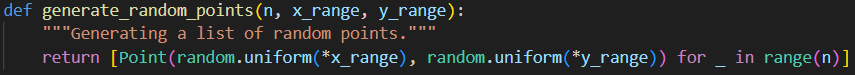
Взагалі, автор вважає алгоритм достатньо перспективним, але він точно потребує подальшої дороботки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Dijkstra E. W. A note on two problems in connexion with graphs (англ.) // Numerische Mathematik / F. Brezzi — Springer Science+Business Media, 1959. — Vol. 1, Iss. 1. — P. 269—271.
2. [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)
3. C. Y. Lee: *An Algorithm for Path Connections and Its Applications*. In: *IRE Transactions on Electronic Computers*. EC-10, Nr. 2, 1961, S. 346–365.

ДОДАТКИ

## ДОДАТОК А.

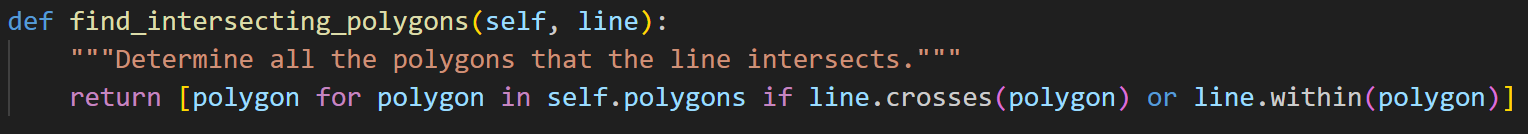


## ДОДАТОК Б.

**Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание**

## ДОДАТОК В.

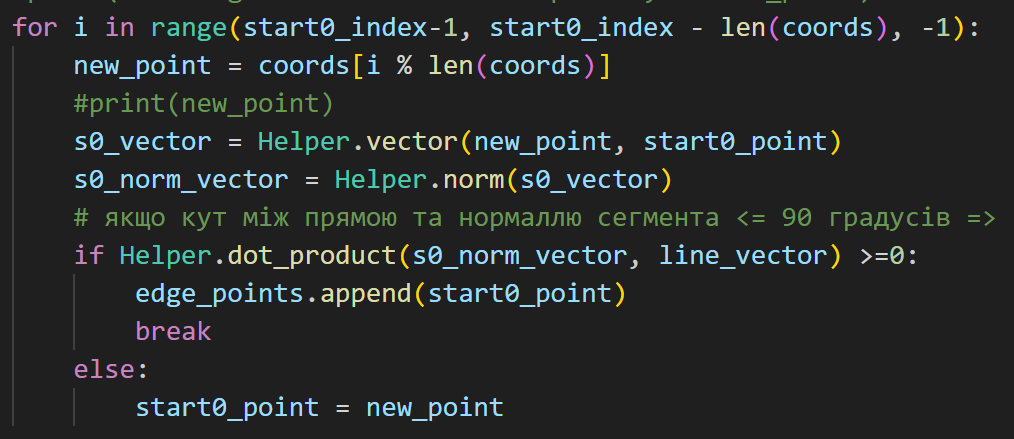
****

## ДОДАТОК Г.

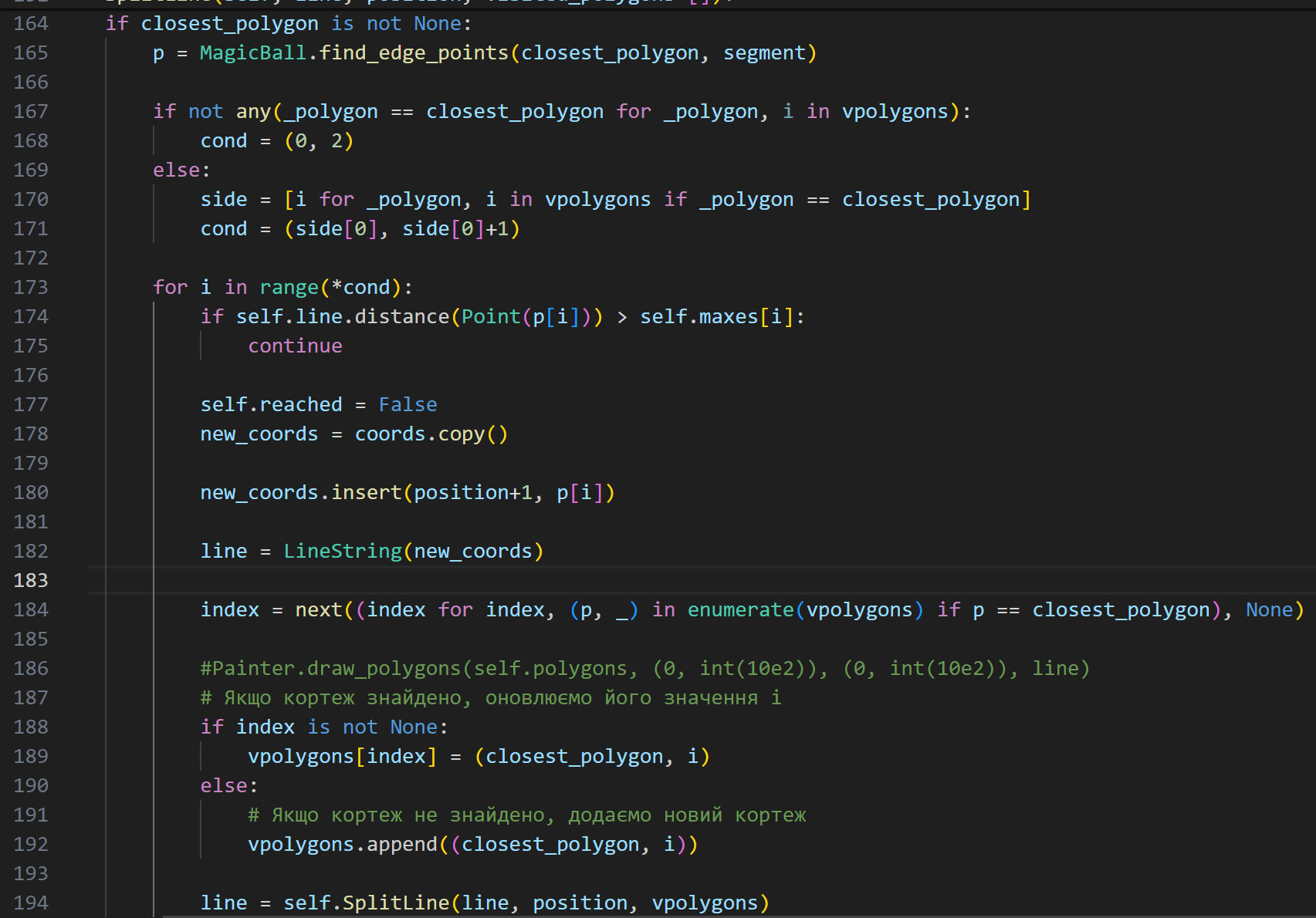
**Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание**

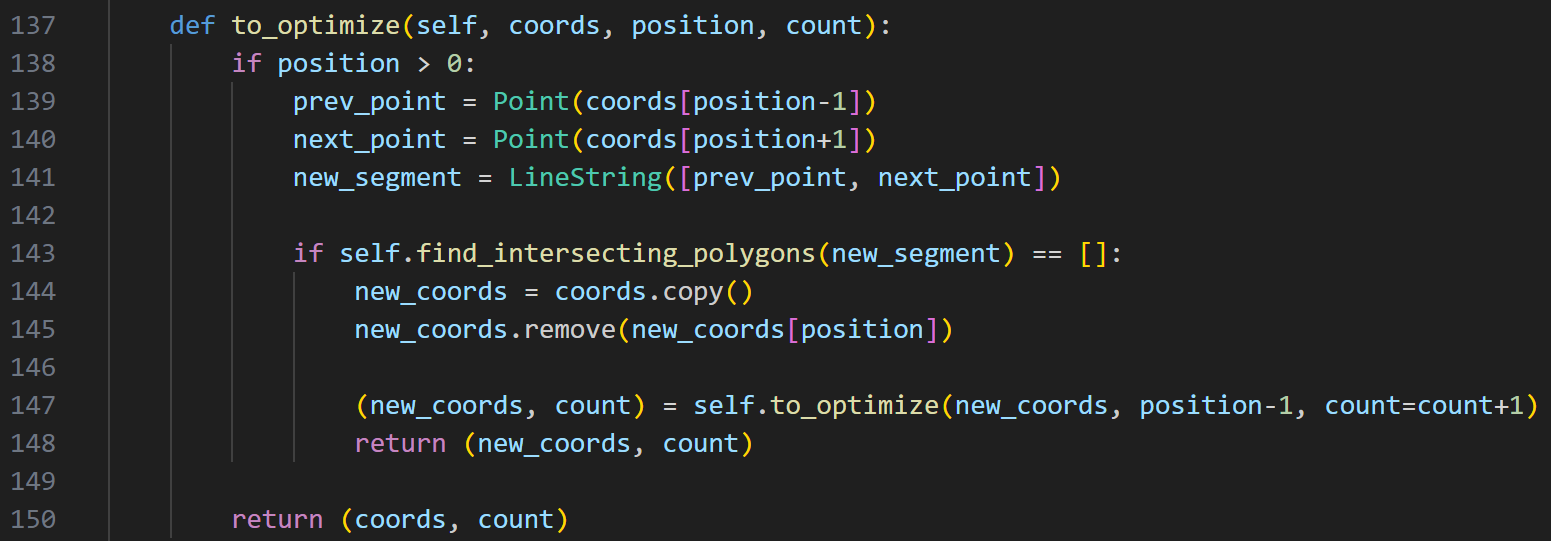
## ДОДАТОК Д.

****

## ДОДАТОК Е

****

## ДОДАТОК Є.

****

## ДОДАТОК Ж.

**Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание**