

1. Локалізація точки на планарному розбитті методом ланцюгів.

У цьому коді реалізовано алгоритм локалізації точки на планарному розбитті методом ланцюгів. Нижче наведено пояснення процесу, що відбувається в кожній частині коду:

1. `main.py`:

- Зчитування вершин, ребер та точки з відповідних файлів.
- Сортування вершин за координатою `y`.
- Ініціалізація списків для збереження вхідних та вихідних ребер для кожної вершини.
- Збалансування ваги ребер вгору та вниз по графу.
- Побудова діаграми з відображенням вершин, ребер та заданої точки.

2. `algorhythm.py`:

- Реалізація різних функцій, необхідних для алгоритму локалізації.
- Наприклад, функції для обрахунку загальної ваги ребер, пошуку лівого найбільшого ребра, сортування ребер за обертанням тощо.

3. `plot.py`:

- Модуль для графічного відображення графу з вершинами, ребрами та точкою.

4. `entity.py`:

- Визначення класів `Point` та `Edge`, які представляють вершини та ребра в графі.

5. `read.py`:

- Модуль для зчитування вершин, ребер та точки з файлів.

Цей код залежить від інших файлів `algorhythm.py`, `entity.py`, `plot.py` та `read.py`, які містять відповідні функції та класи для реалізації алгоритму локалізації методом ланцюгів.

Отже, загальна складність алгоритму може бути оцінена як $O(nm)$, де n - кількість вершин, m - кількість ребер у графі.

4. Регіональний пошук. Метод дерева регіонів.

У цьому коді реалізовано метод дерева регіонів для регіонального пошуку точок у заданій області. Основна ідея полягає в тому, щоб створити ієрархічну структуру дерева, де кожен вузол представляє область, яка містить точки. За допомогою цього дерева можна ефективно знаходити точки, які потрапляють у задану область, уникнувши зайвих перевірок.

Головні етапи реалізації:

1. Будується дерево регіонів шляхом рекурсивного поділу областей на підобласті. Кожен вузол дерева представляє область та містить посилання на свої дочірні вузли або список точок, які потрапляють у цю область.
2. Після побудови дерева можна виконувати регіональний пошук точок. Починаючи з кореневого вузла, перевіряється, чи потрібно шукати точки в даному вузлі. Якщо область повністю міститься в межах вузла, перевіряється, чи є дочірні вузли, і якщо так, рекурсивно викликається пошук в цих вузлах. Якщо жоден з цих випадків не відбувається, перевіряються всі точки у вузлі, і ті, які потрапляють у задану область, додаються до результату.
3. Функція `read_points` зчитує точки з файлу.

4. Функція `draw_points` відображає координати точок та областей на графіку.

Даний код реалізує побудову двійкового дерева пошуку та пошук точок у заданому прямокутному регіоні.

[Відобразити](#)

Складність алгоритму залежить від кількості точок та розмірності простору.

Позначимо n - кількість точок, а d - розмірність простору (у даному випадку $d = 2$).

Побудова двійкового дерева має складність $O(n \log n)$, оскільки при кожному побудові дерево розділяється на дві рівні частини. Побудова виконується для кожного рівня рекурсії, їх всього d , тому загальна складність побудови дерева складає $O(dn \log n)$.

Пошук точок у заданому регіоні відбувається за $O(k + m)$, де k - кількість точок, що задовольняють умові належності до регіону, а m - кількість точок, що повертаються в результаті пошуку. У найгіршому випадку, коли всі точки знаходяться у регіоні, складність буде $O(n)$, оскільки всі точки повинні бути перевірені.

Отже, загальна складність алгоритму буде $O(dn \log n)$ при побудові дерева та $O(n)$ при пошуку точок у регіоні.

6. Найближча пара множини точок на площині – метод «розділяй і владарюй».

Цей код реалізує алгоритм "розділяй і владарюй" для знаходження найближчої пари точок на площині. Основні функції, які використовуються в програмі, включають:

1. `_show_plot(points, p1, p2)`: Функція для графічного представлення точок на площині. Приймає список точок `points` і індекси двох точок `p1` і `p2`, які утворюють найближчу пару. Відображає точки на графіку та показує лінію між вибраними точками.
2. `_sorted_list_split(left_list, right_list, to_split_list)`: Функція для розділення списку `to_split_list` на два підсписки `left` і `right` відповідно до того, в якому списку (з `left_list` або `right_list`) знаходиться кожен елемент `to_split_list`. Повертає ці два підсписки.
3. `distance(p1, p2)`: Функція для обчислення відстані між двома точками `p1` і `p2` на площині.
4. `_nearest_pair(points, x_sorted, y_sorted)`: Основна рекурсивна функція, яка знаходить найближчу пару точок на площині. Приймає список точок `points` і відсортовані списки індексів точок `x_sorted` і `y_sorted` за зростанням координати x і y відповідно. Функція рекурсивно розбиває список точок на дві половини, знаходить найближчі пари в кожній половині і обчислює мінімальну відстань `d` між цими парами. Потім функція шукає точки, які знаходяться в околиці смуги ширини `2d` навколо вертикальної лінії `m_x`, і для кожної точки з лівої половини перебирає точки з правої половини, знаходячи найближчу пару точок між ними. Якщо знайдена відстань між цією парою менша за поточну мінімальну відстань `d`, то вона стає новою мінімальною відстанню `d`.
5. `nearest_pair(points, visualize=False)`: Функція, яка використовує попередню функцію `_nearest_pair` для знаходження найближчої пари точок на площині. Приймає список точок `points` і параметр `visualize`, який визначає, чи потрібно відображати графічне представлення точок. Повертає пару індексів точок та відстань між ними. Якщо параметр `visualize` встановлено на `True`, то також відображається графічне представлення.

В файлі `main.py` наведено приклад використання функції `nearest_pair`. Він створює список точок `points` і викликає функцію `nearest_pair` з цим списком. Результат виводиться на екран. Також в цьому файлі є код, який перебирає всі можливі пари точок та обчислює відстань між ними, щоб перевірити правильність роботи алгоритму.

🔄 Regenerate response

Для оцінки складності алгоритму, давайте розглянемо його кроки:

1. Імпорт модулів та визначення початкових даних мають складність $O(1)$.
2. У функції `nearest_pair` відбувається сортування списків `x` і `y` за координатами x та y відповідно. Складність цього кроку становить $O(n \log n)$, де n - кількість точок.
3. Функція `nearest_pair` виконує рекурсивні виклики, поділяючи список точок на підсписки та знаходячи найближчу пару. Складність цієї функції залежить від кількості точок і може бути оцінена як $O(n \log n)$.
4. У функції `nearest_pair`, після знаходження найближчої пари підсписків, відбувається пошук точок, що знаходяться в околі роздільної лінії та обчислення відстаней між цими точками. Складність цього кроку залежить від кількості точок у роздільних підписах і може бути оцінена як $O(n)$.

Отже, загальна складність алгоритму буде $O(n \log n)$, де n - кількість точок.

A. Динамічна підтримка опуклої оболонки (Overmars, van Leeuwen).

Даний код реалізує анімацію динамічної підтримки опуклої оболонки з використанням бібліотеки `matplotlib`.

У файлі `hull_anim.py`:

1. Створюється початковий стан системи `init_state`, що представляє собою матрицю розміром 10x4. Кожний рядок цієї матриці відповідає одній частинці і містить координати x і y , а також швидості по x і y .
2. Створюється об'єкт `ParticleBox` з використанням початкового стану `init_state` та обмежень `bounds`. Цей об'єкт представляє систему частинок.
3. Встановлюється крок часу `dt` для анімації.
4. Створюється графічний вікно для відображення анімації з використанням `matplotlib`.
5. Налаштовуються межі графічного вікна і маркери для частинок і оболонки.
6. Створюється прямокутник `rect`, який представляє межі системи частинок.
7. Визначаються функції `init()` і `animate(i)`, які використовуються для ініціалізації анімації та оновлення кадрів анімації відповідно.
8. Використовується функція `animation.FuncAnimation` для створення анімації з викликом функції `animate` на кожному кадрі.
9. Виводиться анімація на екран.


У файлі `hull.py`:

1. Визначається набір допоміжних функцій та класів для обчислення опуклої оболонки.
2. Функція `convexHull(points)` приймає список точок `points` і обчислює опуклу оболонку цих точок. Вона використовує алгоритм Грехема для обчислення опуклої оболонки.
3. Оператор `itemgetter` і функція `cmp_to_key` використовуються для визначення порядку сортування точок при обчисленні опуклої оболонки.

У файлі `ParticleBox.py`:

1. Визначений клас `ParticleBox`, який представляє систему частинок.
2. У конструкторі класу встановлюється початковий стан системи, обмеження та розмір частинок.
3. Метод `step(dt)` оновлює стан системи на крок часу `dt`. Він оновлює положення частинок, перевіряє перетин межі системи і змінює швидкості частинок при перетині.

Отже, код створює анімацію системи частинок, в якій виконується рух частинок та будується опукла оболонка цих

 Regenerate response

Найголовніше:

- Складність коду залежить від кількості точок.
- Загальна складність коду залежить від найскладнішого файлу `hull.py` і буде $O(n \log n)$, де n - кількість точок.

Загальна складність коду залежить від найскладнішого файлу `hull.py`, оскільки саме в цьому файлі виконується обчислення опуклої оболонки (convex hull) для заданої множини точок. Обчислення опуклої оболонки вимагає впорядкування точок, визначення їх орієнтації та виконання інших операцій, які мають складність $O(n \log n)$, де n - кількість точок. Тому загальна складність коду залежить від цього найскладнішого етапу, оскільки інші частини коду, такі як анімація часток чи ініціалізація, мають меншу складність.