Algorytmy Geometryczne Lab4
Znajdowanie przecięć odcinków na płaszczyźnie za pomocą algorytmu zamiatania

Nieć Witold

Spis treści

1. Dane Podstawowe

Nazwisko, Imię: Nieć, Witold
<removed>

2. Dane Techniczne

Język: Python 3.12.0
Procesor: Intel core 7 U 155V
System operacyjny: Windows 11
RAM: 16GB
Środowisko: Jupyter Notebook, Pycharm

3. Wstęp

Celem sprawozdania jest analiza i implementacja algorytmu zamiatania (ang. sweep line algorithm) w kontekście znajdowania przecięć odcinków na płaszczyźnie. Algorytm ten pozwala efektywnie wykrywać punkty przecięcia wielu odcinków, co ma zastosowanie w grafice komputerowej, geometrii obliczeniowej i analizie danych przestrzennych.

4. Wprowadzenie teoretyczne

4.1. Opis problemu

Na płaszczyźnie zadanych jest n odcinków, zdefiniowanych przez ich końce (x_1, y_1) i (x_2, y_2). Celem jest znalezienie wszystkich punktów, w których dowolne dwa odcinki się przecinają. Zaliczamy tylko te przecięcia które nie leżą na żadnym z końców obu odcinków.

4.2. Algorytm zamiatania

Algorytm zamiatania polega na przesuwaniu pionowej linii zamiatania (ang. sweep line) od lewej do prawej strony układu współrzędnych. Odcinki aktualnie przecinające zmiotkę są przechowywane w strukturze stanu umożliwiającej szybkie wstawianie, usuwanie oraz znajdowanie sąsiadów.

4.3. Zdarzenia

Zdarzenia to miejsca w których zatrzymuje się zmiotka. Dzielą się na 3 kategorie:

- Początek odcinka
- Koniec odcinka
- Punkt przecięcia

Gdy zmiotka zatrzymuje się, aktualizowana jest struktura stanu (struktura przechowująca wszystkie aktywne odcinki) i sprawdzane są potencjalne przecięcia pomiędzy nowo powstałymi parami sąsiadów.

4.4. Aktywne odcinki

Aktywne odcinki przechowywane są w **strukturze zdarzeń**. Taka struktura musi umożliwiać szybkie wstawianie, usuwanie oraz znajdowanie sąsiadów w celu aktualizacji struktury oraz sprawdzania przecięć. Struktura ta musi być posortowana

4.5. Sprawdzanie przecięć

Gdy dodawany jest nowy odcinek do aktywnej listy, sprawdzaj potencjalne przecięcia z jego sąsiadami w liście.

Gdy odcinek jest usuwany z aktywnej listy, ponownie sprawdzaj przecięcia sąsiadujących odcinków.

Natomiast gdy zmiotka zatrzymuje się w punkcie będącym przecięciem, odcinki w strukturze stanu są zamieniane miejscami i sprawdzane są przecięcia z nowo powstałymi sąsiadami.

5. Struktury danych

5.1. Struktura zdarzeń - kolejka priorytetowa

Zdarzenia przechowywane są w kolejce priorytetowej a kluczem sortowania jest współrzędna X zdarzenia.

Przechowywane są informacje o współrzędnej X zdarzenia, rodzaju zdarzenia oraz indeksach prostych powiązanych z danym zdarzeniem.

5.2. Struktura stanu - drzewo AVL

Implementowana jako samobalansujące się drzewo poszukiwań binarnych.

Złożoność wstawiania, usuwania oraz znajdowania sąsiadów wynosi log n.

AVL to rodzaj zrównoważonego drzewa binarnego wyszukiwania (BST), które dynamicznie utrzymuje swoją wysokość w równowadze. Każdy węzeł w drzewie AVL posiada tzw. balans, który jest różnicą wysokości jego lewego i prawego poddrzewa. Aby drzewo AVL było zrównoważone, wartość balansu każdego węzła musi należeć do zbioru {-1, 0, 1}.

Po każdej operacji wstawiania lub usuwania węzła drzewo AVL dokonuje rotacji (jednej lub kilku), aby przywrócić równowagę. Dostępne rotacje to:

- Rotacja jednokrotna (lewa lub prawa),
- Rotacja podwójna (lewo-prawa lub prawo-lewa).

Drzewo AVL oferuje wydajność wyszukiwania, wstawiania i usuwania rzędu $O(\log \mathrm{n})$

5.3. Pseudokod

5.3.1. Inicjalizacja

Dodaj początki i końce wszystkich odcinków do struktury wydarzeń.

Posortuj zdarzenia według współrzędnej x.

5.3.2. Przetwarzanie wydarzeń

Dla każdego zdarzenia w kolejce: Jeśli zdarzenie to początek odcinka:
- Dodaj odcinek do aktywnej listy.
- Sprawdź przecięcia z sąsiadami.

Jeśli zdarzenie to koniec odcinka:
- Usuń odcinek z aktywnej listy.
- Sprawdź przecięcia między sąsiadami.

Jeśli zdarzenie to przecięcie:
- Dodaj punkt przecięcia do wyniku.
- Dodaj punkt przecięcia do struktury zdarzeń
- Zmień kolejność odcinków w aktywnej liście.
- Sprawdź przecięcia z nowymi sąsiadami

6. Inne struktury danych

6.1. Drzewo BST

W odróżnieniu od drzewa AVL, struktura ta nie posiada mechanizmu automatycznego balansowania, co oznacza, że wysokość drzewa może znacząco wzrosnąć w przypadku niekorzystnych operacji wstawiania lub usuwania. W efekcie, w najgorszym scenariuszu, kiedy drzewo przyjmuje formę zbliżoną do listy (np. w przypadku danych wstawianych w sposób posortowany), złożoność czasowa operacji wstawiania i usuwania elementów może osiągnąć $O(n)$.

6.2. Posortowana lista

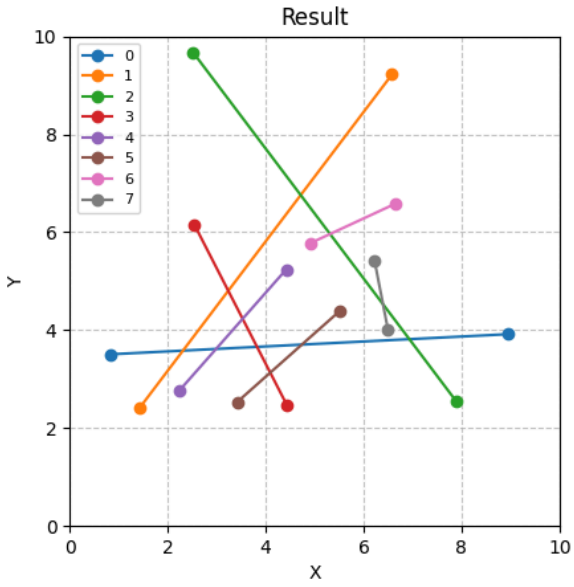
Użyta implementacja struktury posortowanej listy pochodzi z biblioteki *SortedContainers*. Umożliwia ona efektywne wyszukiwanie, wstawianie oraz usuwanie w złożoności $O(\log \mathrm{n})$, podobnie jak drzewo AVL. Wymagane jest, aby klucz używany do porównań elementów był jednoznacznie zdefiniowany, co oznacza, że dla dowolnych dwóch elementów możliwe jest jednoznaczne ustalenie relacji porządku

(mniejszy, większy). Dodatkowo, struktura wymaga, aby nie występowały elementy o identycznych kluczach, czyli klucz porównań musi zapewniać brak duplikatów w zbiorze.

7. Testy

7.1. Zestawy testowe

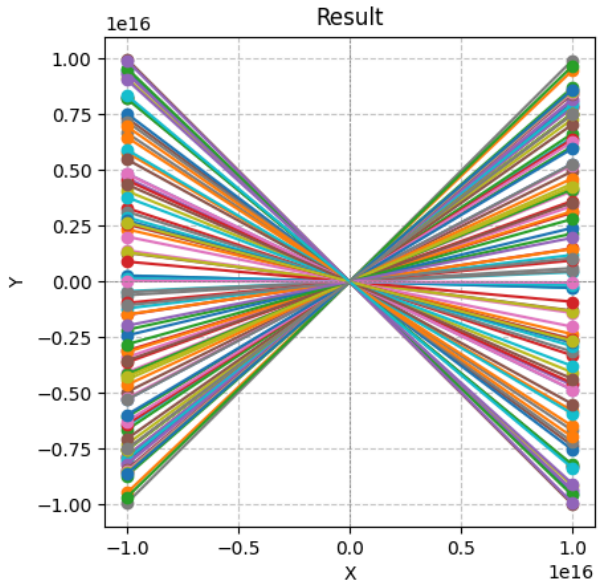
7.1.1. Zestaw A



Rysunek 1: Zestaw A

Zestaw A to zestaw testowy. Został on uwzględniony aby pokazać poprawność działania algorytmu. Jedyną trudnością może być wykrycie przecięć prostych 2 i 7 ze względu na ich duże nachylenia.

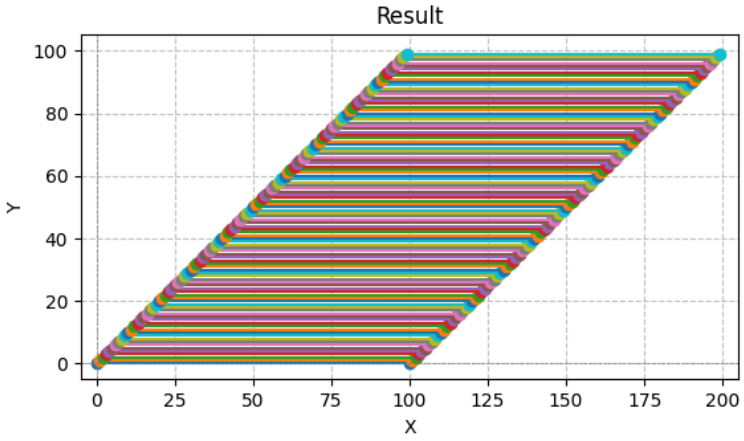
7.1.2. Zestaw B



Rysunek 2: Zestaw B

Zestaw B sprawdza działanie algorytmu w przypadku gdy wiele odcinków przecina się w jednym punkcie. Trudność może sprawić zarządzanie strukturą stanu w punkcie przecięcia oraz powielanie punktów przecięć

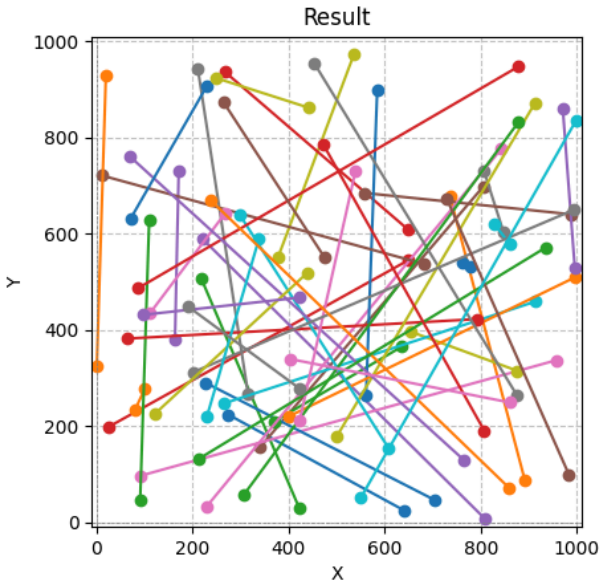
7.1.3. Zestaw C



Rysunek 3: Zestaw C

Zestaw C to zbiór poziomych prostych, równoległych. Ze względu na ich ułożenie, nie powinno zostać wykryte żadne przecięcie. W tym przypadku testowana jest efektywność rotacji drzewa AVL ponieważ odcinki podawane są w kolejności posortowanej od góry do dołu - przy każdym dodawaniu i usuwaniu ze struktury konieczne będą rotacje drzewa.

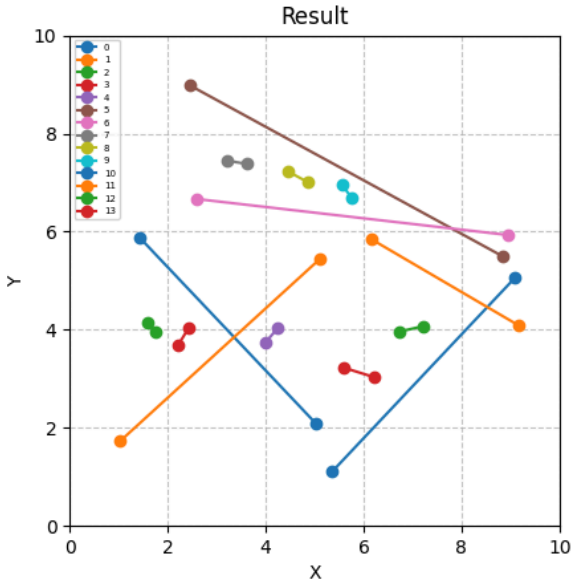
7.1.4. Zestaw D



Rysunek 4: Zestaw D

Zestaw D to zbiór losowo wygenerowanych odcinków w kwadracie o wymiarach 1000 x 1000. Testowana jest głównie wydajność oraz dokładność algorytmu.

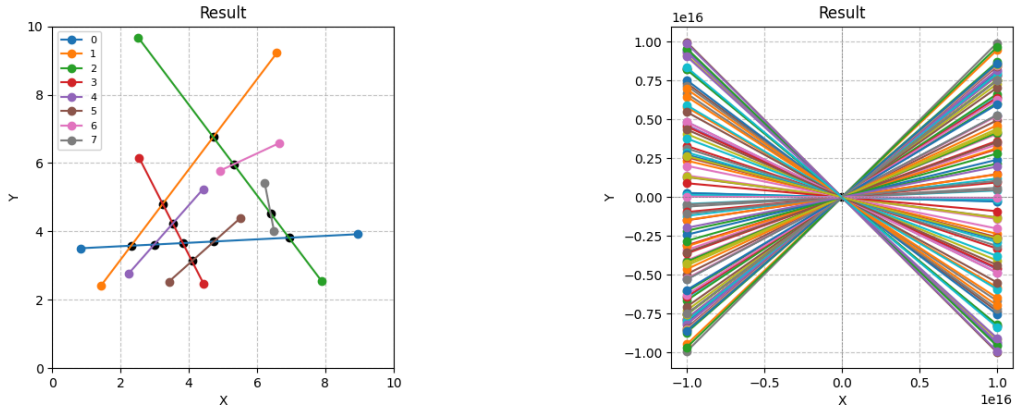
7.1.5. Zestaw E



Rysunek 5: Zestaw D

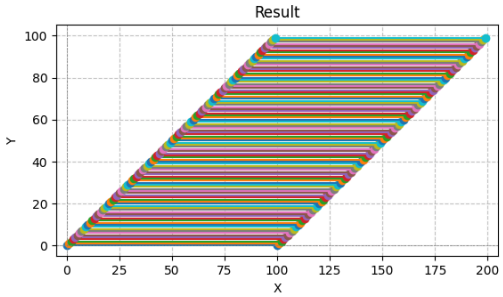
Zestaw E zawiera taki układ odcinków by każde z przecięć było wykrywane więcej niż jeden raz. Poprzez wstawienie mniejszych odcinków pomiędzy proste które się przecinają, wymuszamy kilkukrotne sprawdzanie przecięcia.

7.2. Wyniki algorytmu



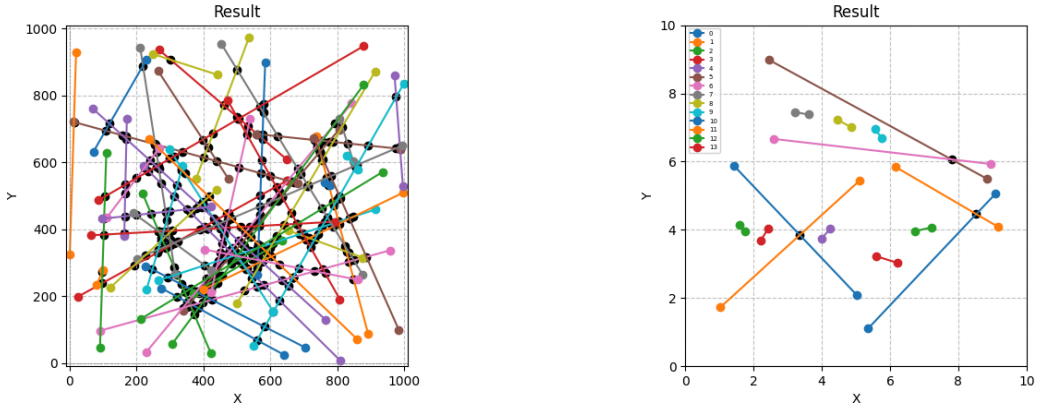
Rysunek 6: Wyniki algorytmu dla zestawów A i B

Jak widać na rysunkach powyżej, dla obu zestawów znaleziono poprawnie wszystkie punkty przecięć. Dla zestawu B poprawnie został znaleziony tylko jeden punkt.



Rysunek 7: Wynik algorytmu dla zestawu C

Dla zestawu C algorytm zgodnie z oczekiwaniami nie znalazł żadnego punktu przecięcia. Wszystkie odcinki są równoległe i nie mają punktów wspólnych.



Rysunek 8: Wyniki algorytmu dla zestawów D i E

Ciężko stwierdzić czy dla zestawu D, wszystkie przecięcia zostały wykryte. Natomiast dla zestawu E można łatwo zobaczyć że pomimo wielokrotnego wykrywania przecięć przez algorytm, udało się znaleźć wszystkie oczekiwane punkty.

7.3. Porównanie efektywności zaimplementowanych struktur danych

7.3.1. Zestaw B

Wydajność obu struktur dla zestawu B jest zbliżona, ze wskazaniem na drzewo AVL. Dla coraz większej liczby prostych widzimy lekką przewagę drzewa AVL ze względu na stałą złożoność logarytmiczną wyszukiwania odcinków.

7.3.2. Zestaw C

Jak można było by się spodziewać, zestaw C ukazuje wady drzewa BST. Dla odcinków podawanych w kolejności posortowanej, z drzewa BST tworzy się struktura przypominająca listę. Przez to operacje wyszukiwania przyjmują złożoność liniową

7.3.3. Zestaw D

Zestaw D pokazuje że drzewo AVL jest bardziej odpowiednie niż zwykłe drzewo BST dla algorytmu zamiatania. Ponieważ zapewnia równoważenie drzewa, co przekłada się na lepszą wydajność w operacjach związanych z przeszukiwaniem, wstawianiem i usuwaniem elementów. Dodatkowo lepiej sprawdza się dla algorytmów wymagających dynamicznych zmian w strukturze.

7.4. Wnioski

Algorytm zamiatania poprawnie identyfikuje punkty przecięć odcinków we wszystkich testowanych przypadkach, w tym w zestawach z odcinkami przecinającymi się wielokrotnie oraz równoległymi, co świadczy o jego skuteczności.

Struktura AVL okazała się bardziej wydajna w porównaniu z BST, szczególnie w przypadkach, gdzie wymagana była duża liczba operacji przeszukiwania, wstawiania i usuwania. W zestawach z uporządkowanymi danymi BST wykazuje znaczną degradację wydajności.

Testy dla zestawu D (losowe odcinki) oraz zestawu E (wielokrotne przecięcia) pokazały, że algorytm dobrze radzi sobie w sytuacjach wymagających dynamicznych zmian w strukturze stanu.

Algorytm zamiatania ma szerokie zastosowanie w geometrii obliczeniowej i analizie przestrzennej, co czyni go cennym narzędziem w aplikacjach takich jak grafika komputerowa i systemy GIS.