Algorytmy Geometryczne

Sprawozdanie z ćwiczenia 4. „Przecinanie się odcinków”

Maciej Wiśniewski
Grupa 3 Poniedziałek 16.45 A

Data wykonania 10.12.2024 Data oddania 13.12.2024

1. Dane techniczne

Specyfikacja komputera: system Ubuntu 24.04.01 Linux 5.15 x64, procesor AMD Ryzen 7 5825U with Radeon 2GHz 8 rdzeni, 16GB pamięci RAM. Ćwiczenie zostało napisane w języku ***Python 3.9.20*** w ***Jupyter Notebook*** w środowisku programistycznym ***Visual Studio Code.*** Aby wykonać ćwiczenie posłużono się bibliotekami: ***numpy matplotlib, sortedcontainers, PIL, json*** i ***tkinter*** . Do wykonania wizualizacji stworzono specjalne klasy oraz funkcje, które później zostaną szczegółowo opisane.

2. Cel ćwiczenia

Celem ćwiczenia była implementacja algorytmów wyznaczających pierwsze i wszystkie przecięcia punktów na płaszczyźnie przy pomocy algorytmu ***zamiatania***(*sweep line algorithm*). Dodatkowo ćwiczenia obejmowało wizualizację i analizę wyników.

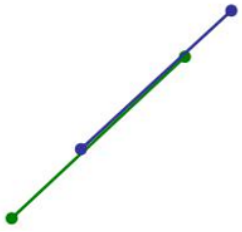
3. Wstęp teoretyczny

Algorytm zamiatania oparty jest o wykorzystanie ***„miotły”***, prostej równoległej do osi ***OY.***
Aby rozwiązać problem przecinania się odcinków na płaszczyźnie posługując się algorytmem ***zamiatania*** musimy poczynić następujące pewne założenia:

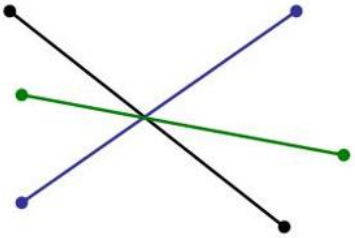
I. Żaden inny rozważany odcinek jest równoległy do osi ***OY***
II. Dwa odcinki przecinają się w co najwyżej jednym punkcie
III. W jednym punkcie przecinają się nie więcej niż dwa odcinki

Następujące sytuacje są niemożliwe: Rysunek 1 – ze względu na warunek I., Rysunek 2 – ze względu na warunek II., Rysunek 3 – ze względu na warunek III.

Rysunek 1 Odcinek równoległy do osi OY

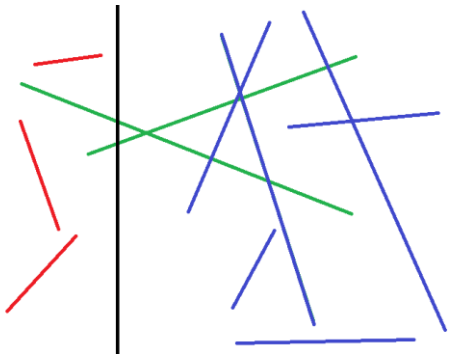


Rysunek 2 Odcinki przecinające się w więcej niż jednym punkcie



Rysunek 3, Więcej niż dwa odcinki przecinające się w jednym punkcie

„Miotła” przechodzi od ujemnych współrzędnych na osi ***OX*** w stronę dodatnich, z II i III ćwiartki układu współrzędnych do I i IV, i rozdziela zbiór odcinków na trzy podzbiory: przetworzone, aktywne i oczekujące(Rysunek 4). Zamiatanie odbywa się w tym samym kierunku. Miotła zatrzymuje się w punktach zdarzeń, którymi są końce odcinków oraz wykryte punkty przecięć. To właśnie tutaj następuje aktualizacja stanu miotły oraz testy przecięć.



Rysunek 4, Miotła i zbiory punktów przetworzonych, aktywnych i oczekujących

Stanem miotły w algorytmie ***zamiatania*** nazywamy dynamicznie utrzymywaną strukturę, tak zwaną strukturę ***strukturę stanu***, oznaczą przez ***T***. W tej strukturze przetrzymywane są analizowane odcinki, czyli te oznaczana jako aktywne. Ta struktura przechowuje ważne informacje w konekście obliczeń.
Dodatkowo algorytm przetrzymuje jeszcze jedną strukturę pomocniczną, ***strukturę zdarzeń***, oznaczaną przez ***Q***. Ta struktura przechowuje zdarzenia, które są zdefiniowane akutalnie rozważanym punkcie. Jako zdarzenie rozpoznajemy jako jeden z następujących stanów: „początek odcinka”, „przecięcie odcinków” lub „ koniec odcinka”.

4. Implementacja algorytm i wizualizacji

Aby zwizualizować działanie algorytmu zaimplementowano na początku zaimplementowano interaktywną funkcję do podawania odcinków poprzez rysowanie ich na płaszczyźnie. Implementację tej funkcji oparta na bibliotece ***matplotlib***. Dodatkowo dodano funkcje umożliwiające wczytywanie odcinków z plików zewnętrz jak również wpisywanie współrzędnych początku i końca odcinka ręcznie.
Następnie zdefiniowano klasy pomocnicze: ***Point, Line, Point_Coll, Line_Coll***, których zadaniem jest generowanie i przechowywanie obiektów(punktów i odcinków). Do przechowywanie pojedynczych klatek programu stworzono klasę ***Scene***, które wraz z klasą ***_Button_callback*** stanowią podstawę dla głównej klasy odpowiedzialnej za interaktywne menu, służące do przeglądania kroków programu, tworzone przez klasę ***Plot***. Na koniec dodano funckję ***create_sweep_gif,*** która w opraciu o zdefiniowane wcześniej klasy, prezetuje automatyczne pliki ***.gif***. Pliki z reprezentajcą krokową algorytmu w postaci plików ***gif*** przedstawiające zarówno algorytm znajdujący pierwsze przecięcie jak i wszystkie przecięcia odcinków na płaszczyźnie dostępne są w oddzielnym pliku.

Przed rozpoczęciem algorytmu należało rozważyć w jaki sposób optymlanie zaimpelmentować struktury ***zdarzeń*** i ***stanu***. Dla struktury ***stanu*** wybrałem ***SortedSet*** dostępny w bibliotece ***sortedcontainers***, a dla struktury ***zdarzeń PriorityQueue***.

SortedSet to struktura danych reprezentująca zbiór elementów w posortowanym porządku, która wspiera szybkie operacje dodawania, usuwania i wyszukiwania w czasie ***O(logn)***. Wykorzystuje ona drzewa ***BST*** do dynamicznego utrzymywania kolejności elementów. W ***SortedSecie*** łatwo i efektywnie można przeszukiwać sąsiadów, czyli poprzednika i następnika danego elementu w w czasie ***O(logn).*** Dodatkowo ***SortedSet*** pozwala na dynamiczne i efektywne zarządzanie zmianami porządku elementów w liście. Dużą zaletą ***SortedSeta*** jest też jego łatwość w obsłudze.

PriorityQueue jest dobrym wyborem dla struktury ***zdarzeń***, ponieważ zapewnia przetwarzanie w ścisłej kolejności, dodatkowo jest efektywna w tej obsłudze. Umożliwia ona dodawanie nowych zdarzeń i pobieranie zdarzenia o najwyższym priorytecie w czasie ***O(logn).*** Jest ona dobrze znamym i łatwym w obsłudze narzędziem, co ułatwia zrozumienie kodu, w którym została użyta.

Algorytm wraz z wizualizacją

T – struktura stanu
Q – struktura zdarzeń

Wersja algorytmu szukająca pierwszego przecięcia nieznacznie różni się od głównego algorytmu szukające wszystkie przecięć odcinków na płaszczyźnie. W obu implementacjach użyto tych samych struktur.
Na początku tworzymy sktuktury ***zdarzeń*** i ***stanu***, dodatkowy słownik ***used***, który wyeliminuje nam powtórzenia, tablice na wyniki, klasyfikujemy punkty do ***struktury zdarzeń,*** dopóki **Q** nie jest puste, wyciągamy punkt z kolejki i rozważamy jakie zdarzenie reprezentuje:
- jeśli punkt jest porządkiem odcinka, dodajemy odcienk do ***T***, szukamy jego sąsiadów i sprawdzamy czy mogło nastąpić przecięcie, jeśli tak dodajemy nowe zdarzenie do ***Q,***
- jeśli punkt jest końcem odcinka, usuwamy go z ***T*** i sprawdzamy czy „nowi" sąsiedzi w ***T*** się przecinają,
- jeśli punkt jest przecięciem usuwamy przecinające się odcinki z ***T***, przestawiamy miotłę, ponownie wstawiamy dwa poprzednio usunięte odcniki i sprawdzamy ich nowe relacje z sąsiadami.
Po każdym zdarzeniu wizualizowana jest scena z aktualnym stanem miotły i przecięciami.

Korzyści takiego podejścia algorytmicznego

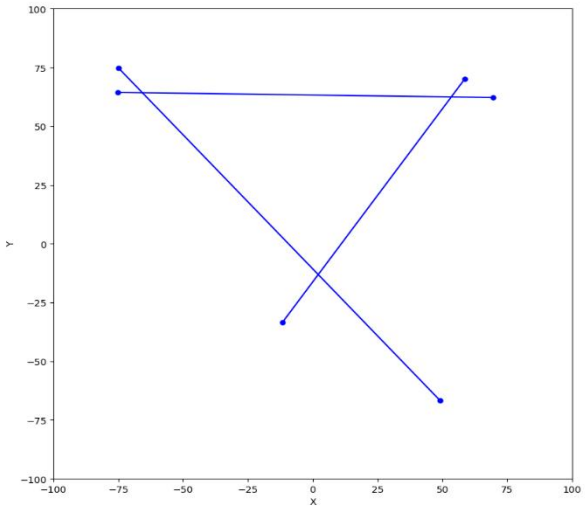
- Wysoka wydajność, złożoność powyższego algorytmu to O((***n+k)logn***), gdzie *n* to liczba odcinków, a ***k*** to liczba przecięć, dla małych ilości przecięć jest znacznie szybszy od typowego podejścia o złożoności O($\boldsymbol{n^2}$)
- Dynamicznie zarządzanie przecięciami, algorytm obsługuje przecięcia w locie, dodając je do kolejki priorytetowej w czasie rzeczywistym, co pozwala efektywnie wykrywać nawet skomplikowane konfiguracje przecięć,
- Uniwersalność, podany algorytm można stosować do różnych problemów geometycznych(na przykładzie algorytm zamiatania jest użytwany do konstrukcji ***Wieloboków Voronoi*** przy użyciu algorytmu ***Fortune'a***)
- Efektywność, użycie struktury ***SortedSet*** (do stanu miotły) i kolejki priorytetowej (do zdarzeń) minimalizuje liczbę operacji na odcinkach, dzięki czemu algorytm analizuje tylko istotne pary odcinków.
- Niska złożoność pamięciowa O(***n+k***)
- Dla tego algorytmu można też łatwo przedstawić wizualizację.

5. Testowanie

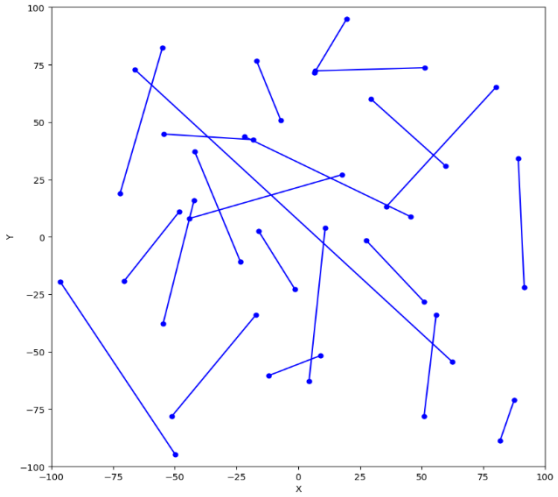
W celu przetestowanie działania algorytmu przygotowano zbiory danych, które mogą pokazać działanie programu w wielu granicznych przypadkach:

- **Zbiór 1** – układ trzech odcinków wzajemnie się przecinających (Rysunek 5), sprawdzenie poprawności działania algorytmu dla przypadków elementarnych,
- **Zbiór 2** – losowe rozmieszczenie odcinków (Rysunek 6), działanie algorytmu w sytuacja nieprzewidywalnych i dla dużej liczby przecięć,
- **Zbiór 3** – układ bez przecięć (Rysunek 7), sprawdzenie czy algorytm poprawnie nie wykryje żadnego przecięcia,
- **Zbiór 4** – test sprawdzający czy algorytm poprawnie zachowa się w sytuacji, kiedy potencjalnie może kilkukrotnie znaleźć punkt przecięcia (Rysunek 8),
- **Zbiór 5** – wszsytkie przecięcia znajdują się tylko w małej podprzestrzeni płaszczyzny(Rysunek 9),
- **Zbiór 6** – zbiór, w którym jeden odcinek przecina się ze wszystkimki innymi na płaszczyźnie(Rysunek 10), sprawdzanie czy algorytm poprawnie będzie przetrzymywał odcienk na miotle jeśli znajedzie dla niego kilka przecięć,
- **Zbiór 7** – przetrzymywanie wszystkich odcinków na miotle(w pewnym momencie) (Rysunek 11), sprawdzenie czy algorytm odpowiednio przechowuje elementy w strukturze,
- **Zbiór 8** – zbiór, w którym bierzemy dwa punkty, znajdujące się dokładnie w tym samym miejscu (Rysunek 12), ten test sprawdza dokładność obliczeń, czy algorytm poprawnie sklasyfikuje punkt(nieskończenie krótki odcinek) jako pionowy, czy znajdzie przecięcie jeśli dwa nieskończenie krótkie odcinki się przetną.

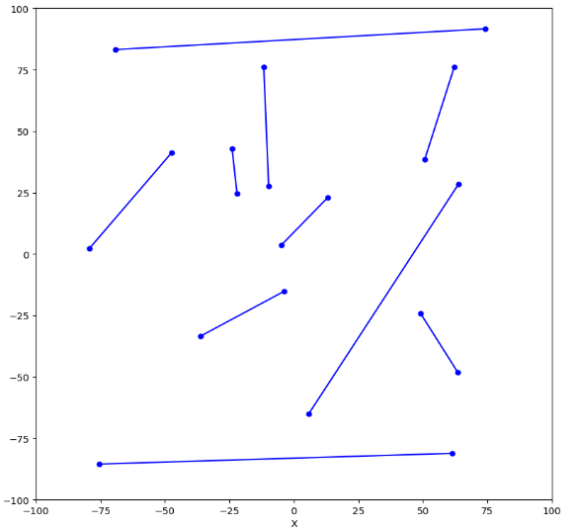
Wyniki testów przedstawiono w tabeli (Tabela 1).



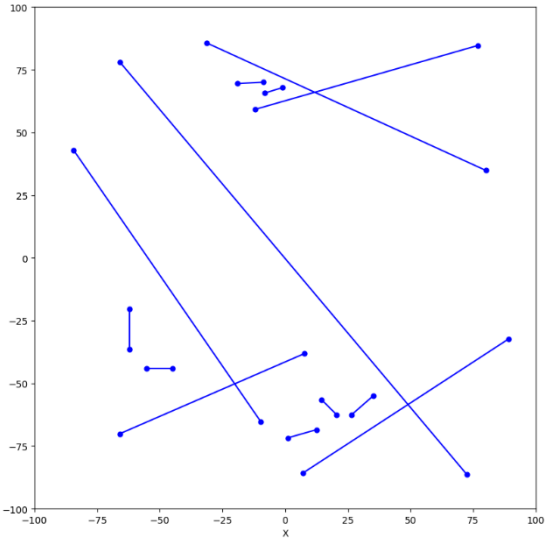
Rysunek 5 Wizualizacja Zbioru 1



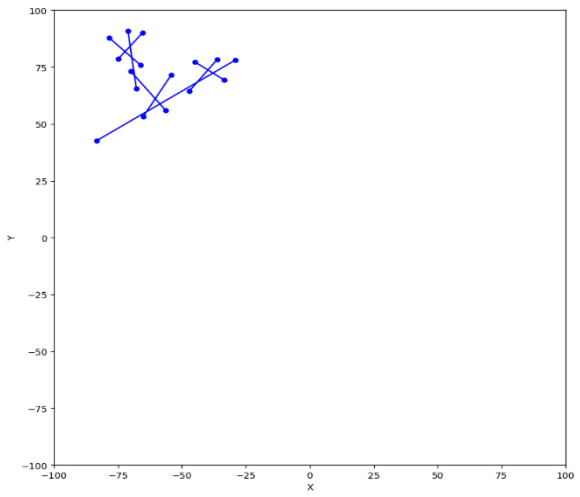
Rysunek 6 Wizualizacja Zbioru 2



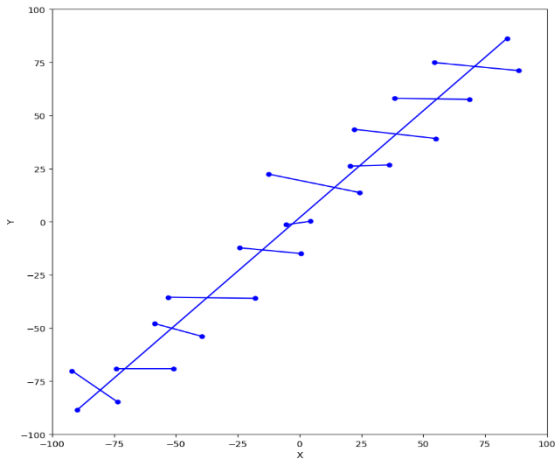
Rysunek 7 Wizualizacja Zbioru 3



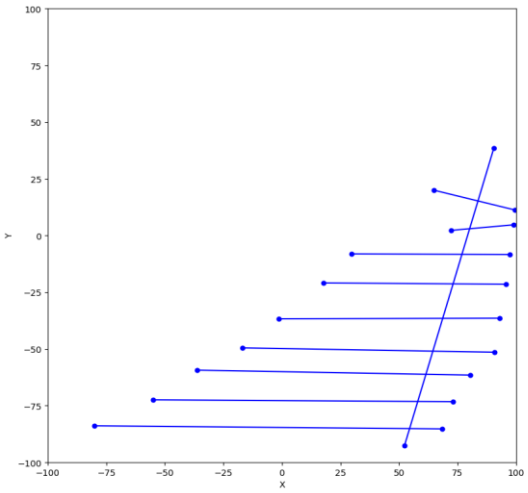
Rysunek 8 Wizualizacja Zbioru 4



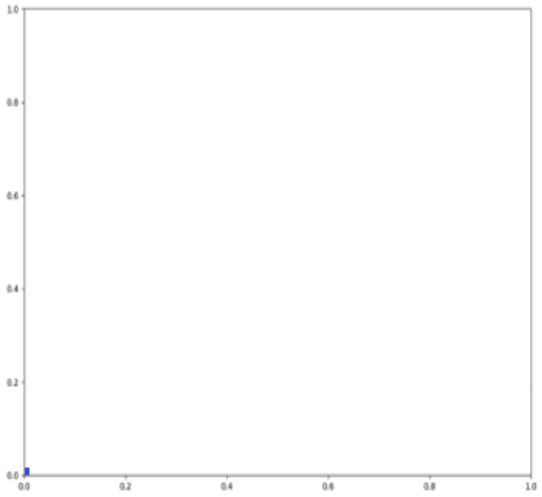
Rysunek 9 Wizualizacja Zbioru 5



Rysunek 10 Wizualizacja Zbioru 6



Rysunek 11 Wizualizacja Zbioru 7



Rysunek 12 Wizualizacja Zbioru 8

	Oczekiwana	Otrzymana
Zbiór 1	3	3
Zbiór 2	13	13
Zbiór 3	0	0
Zbiór 4	3	3
Zbiór 5	10	10
Zbiór 6	11	11
Zbiór 7	9	9
Zbiór 8	0	0

Tabela 1.
Przedstawienie wyników działania algorytmu dla Zbiorów 1-8, wartość oczekiwana została policzona metodą brute-force, algorytmem o złożoności obliczeniowej O($\boldsymbol{n^2}$).

Analizując wyniki przedstawione w Tabeli 1, zaimplementowana wersja algorytmu działa poprawnie.

6. Podsumowanie i wnioski

W sprawozdaniu przedstawiono implementację algorytmu ***zamiatania*** (sweep line algorithm) służącego do wykrywania punktów przecięć odcinków na płaszczyźnie. Zasugerowana implementacja oparta na podejściu obiektowym z użyciem ***SortedSet*** i ***PriorityQueue*** przeszła wszystkie testy sprawdzające zarówno przypadki trywialne jak i sytuacje brzegowe. Program przeszedł również testy zasugerowane przez ***Koło Naukowe BIT***. Program bazował na algorytmie przedstawionym na wykładzie. Wysoka efektywność obliczeniowa i niska złożoność pamięciowa czynią program doskonałym rozwiązaniem dla problemu przecięć odcinków na płaszczyźnie. Mimo, że w sytuacjach, kiedy liczba przecięć zbliża się do kwadratu ilości odcinków lepsza wydaje się metoda brute-force o złożoności O($\boldsymbol{n^2}$), to w zdecydowanej większości przypadków funkcjonalnych, zaprezentowany algorytm będzie o wiele lepszy.