Algorytmy Geometryczne

Sprawozdanie – Ćwiczenie 4: Przecinanie się odcinków

Krzysztof Chmielewski

Data wykonania: 18.11.2024
Data oddania: 02.12.2024

Spis treści

WSTĘP

CEL ĆWICZENIA

Celem tego ćwiczenia jest zapoznanie się z algorytmem wyznaczającym przecięcia odcinków na dwuwymiarowej przestrzeni, a także wykorzystanie odpowiednich struktur potrzebnych do implementacji tego algorytmu, dzięki, którym jego złożoność czasowa będzie niższa.

TEORIA

ZAŁOŻENIA

Podczas realizacji algorytmu zostaną pominięte następujące przypadki: odcinków pionowych, odcinków nałożonych na siebie oraz przypadek, gdy trzy lub więcej odcinków przecina się w jednym punkcie.

OPIS ALGORYTMU

Algorytm opiera się o tzw. zamiatanie. Wyznaczamy prostą (miotłę), która będzie przesuwać się w wyznaczonym kierunku zamiatania zgodnie z określonymi **zdarzeniami**, które będą przetrzymywanie w **strukturze zdarzeń**. Jednocześnie deklarujemy **strukturę stanu**, która będzie przetrzymywać informację potrzebne do wykonywania obliczeń, będzie ona aktualizowana przy każdym zdarzeniu.

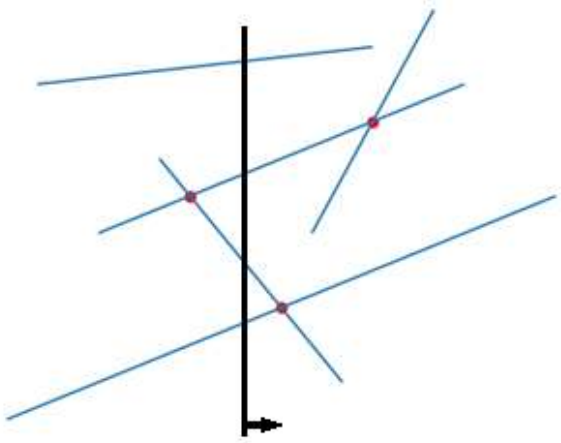
W naszym przypadku miotła będzie przesuwać się w kierunku rosnących wartości x. Miotła będzie zatrzymywać się kolejno w punktach zdarzeń, które będziemy wyciągać ze struktury zdarzeń. Wszystko co jest na lewo od miotły uznajemy za sprocesowane, natomiast to co po prawej będziemy analizować.

Odcinki należy wpierw uporządkować zgodnie z kryterium podanym na wykładzie i zrealizować kroki algorytmu zamiatania.

OPIS WYKORZYSTANYCH STRUKTUR

Struktura zdarzeń Q – zawiera uporządkowane końce odcinków rosnąco w kierunku x-ów. Zawiera także punkty przecięć aktywnych odcinków.

Struktura zdarzeń T – jest to zbiór odcinków aktywnych odpowiednio uporządkowanych względem współrzędnych y dla określonego położenia miotły



Rys. 1 Ilustracja miotły w opisanym algorytmie

Na Rys. 1 pokazano ilustrację miotły w dwuwymiarowej przestrzeni.

DANE TECHNICZNE

Ćwiczenie zostało wykonane z użyciem narzędzia graficznego dostarczonego przez Koło Naukowe Bit (**https://github.com/aghbit/Algorytmy-Geometryczne)**, które umożliwia wizualizację wykresów i kształtów geometrycznych wykorzystujące różne biblioteki języka Python (np. numpy, pandas, matplotlib) oraz Anacondę do stworzenia odpowiedniego jądra dla Jupyter Notebook. Kod zawarty w pliku ***chmielewski_kod_4.ipynb*** został napisany w języku Python właśnie przy użyciu Jupyter Notebook.

Obliczenia zostały wykonane z wykorzystaniem sprzętu o następujących parametrach:

Komputer -> wirtualna maszyna VirtualBox:

- **Procesor:** AMD Ryzen 5 5600X 6 rdzeniowy, podstawowe taktowanie: 3,70 GHz -> w wirtualnej maszynie wykorzystano jedynie 5 z 12 wątków
- **Karta Graficzna:** NVIDIA GeForce RTX 3060 Ti 8GB GDDR6X
- **RAM:** 32GB DDR4 3000MHz CL16 -> w wirtualnej maszynie wykorzystano jedynie 16GB
- **OS:** Linux Ubuntu 24.04

Laptop:

- **Procesor:** Intel Core I5-1235u 10 rdzeniowy, podstawowe taktowanie: 1,30 GHz
- **Karta Graficzna:** zintegrowana z procesorem Intel UHD
- **RAM:** 8GB DDR4 3200MHz
- **OS:** Linux Ubuntu 24.04

REALIZACJA ĆWICZENIA

GENEROWANIE ZBIORÓW ODCINKÓW

Przy realizacji podanego algorytmu mamy dwie opcje wprowadzania zbiorów, na których będą przeprowadzane testy: losowe generowanie zbioru odcinków lub ręczne wprowadzanie odcinków.

Dla generowania losowego zbioru została utworzona funkcja generate_uniform_sections(), która zwraca zbiór losowo wygenerowanych odcinków zgodnie z założeniami, jako argumenty funkcja przyjmuje maksymalną współrzędną x i y, która może zostać wylosowana, a także ilość odcinków do wygenerowania.

Do ręcznego wprowadzania odcinków została napisana funkcja add_sections(), która otwiera okno, w którym można rysować odcinki poprzez zaznaczenie dwóch punktów w przestrzeni. Wybranie jednego punktu sprawia, że algorytm oczekuje na wprowadzenie drugiego punktu jako drugiego końca odcinka, jego wprowadzenie od razu łączy dwa punkty w odcinek. Wybranie punktu wprowadza się za pomocą **lewego przycisku myszy**, aby zaakceptować narysowany zbiór należy kliknąć **spację**. Istnieje także możliwość zresetowania całego obszaru rysowniczego za pomocą przycisku **r**.

WYKORZYSTANE STRUKTURY

W przedstawionym algorytmie wykorzystuję dwie struktury: punktu – Point oraz odcinka – Segment. Struktura punktu jest mało potrzebna z punktu widzenia algorytmicznego, ponieważ w jej miejsce można było wykorzystać krotki, lecz jest ona pomocna, do ujednolicenia kodu i jego czytelności.

- Klasa Point określa jeden punkt w przestrzeni dwuwymiarowej, posiada atrybuty x i y, które są odpowiednikami współrzędnych tego punktu na płaszczyźnie kartezjańskiej. Posiada konstruktor i metody przeciążające operator (==) oraz (>), aby można było określić zależność między dwoma punktami. Posiada także metodę hash, która odpowiednio hashuje atrybuty klasy.
- Klasa Segment określa odcinek między dwoma punktami, posiada atrybuty w postaci punktów końcowych tego odcinka, są one klasy Point, a także jednoznacznie określające prostą parametry alpha i beta, czyli współczynnik kierunkowy prostej i wyraz wolny. Podobnie jak klasa opisana wyżej, Segment posiada metody przeciążające oraz metodę hashującą.

Metody przeciążające są potrzebne do określenia na jakiej podstawie odcinki i punkty są porównywalne, metody hashujące są wymagane do później opisanej struktury SortedSet.

Klasa Segment dodatkowo posiada metodę statyczną, która jest określona dla całej klasy, a nie jedynie obiektu danej klasy, pomaga ona zaznaczyć współrzędną x-ową, na której obecnie znajduje się miotła, dla wszystkich odcinków, a nie tylko jednego.

IMPLEMENTACJA STRUKTURY ZDARZEŃ I STANU

Zarówno struktura zdarzeń Q jak i struktura zdarzeń T zostały zrealizowane jako obiekty klasy SortedSet, która jest częścią zewnętrznej biblioteki sortedcontainers. Dlaczego taki wybór? Klasa SortedSet to mutowalny, posortowany zbiór, którego metody są realizowane w czasie O(logn). Jest to klasa łącząca w sobie cechy zbioru, zatem nie zezwala na powtarzanie się elementów, oraz klasy SortedList, która utrzymuje porządek elementów, a także zapewnia pobieranie elementów, usuwanie ich w czasie O(logn), podobnie operacja dostawania się do indeksu konkretnego elementu zachowuje ten czas.

Jak zainstalować bibliotekę sortedcontainers w swoim środowisku conda?

1. Należy aktywować środowisko poleceniem **conda activate (nazwa_środowiska)**
2. Należy zainstalować bibliotekę za pomocą komendy pip: **pip install sortedcontainers**
3. Można zweryfikować czy biblioteka ta została dodana do środowiska poprzez: **pip list** albo **conda list**

ZMIANY W STRUKTURACH

Do realizacji były dwa algorytmu: pierwszy – sprawdzający czy istnieje przecięcie w danym zbiorze, drugi – wyznaczający wszystkie przecięcia.

Ze względu na to, że najpierw w moim przypadku został zaimplementowany drugi algorytm znajdowania wszystkich przecięć, to nie zmieniałem struktur stanu i zdarzeń. W przypadku pierwszego algorytmu stwierdzenia czy istnieje przecięcie, nie musimy dokładać do struktury stanu następnych przecięć, a jedynie zwrócić True. Alternatywnie można zamiast SortedSet użyć implementacji listowej z sortowaniem, więc dla struktury zdarzeń Q posortowanie ma złożoność czasową O(nlogn), podobnie dlatego, że natrafiając na przecięcie jedynie zwracamy True, a nie zmieniamy kolejności i sąsiadów w strukturze stanu T, wstawianie do posortowanej listy następnego aktywnego odcinka ma złożoność O(n).

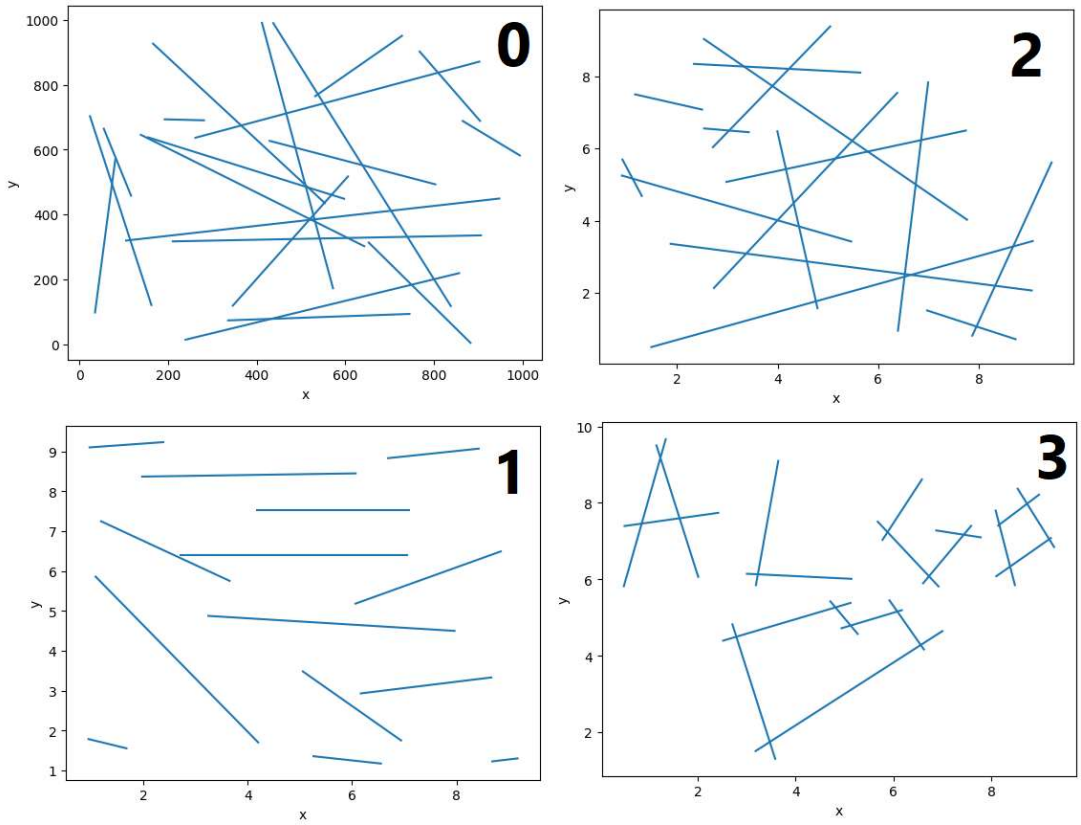
OBSŁUGA STRUKTURY ZDARZEŃs

Zdarzenia początku, końca odcinka oraz przecięcia odcinka są obsługiwane zgodnie z procedurami opisanymi na wykładzie:

- zdarzenie początku odcinka **L** - aktualizujemy położenie miotły, wstawiamy odcinek do struktury stanu T i sprawdzamy sąsiadów danego odcinka, jeśli znajdujemy jakieś przecięcie to wstawiamy je do struktury zdarzeń Q
- zdarzenie początku odcinka **R** - aktualizujemy położenie miotły, usuwamy odcinek ze struktury stanu T i sprawdzamy czy sąsiedzi tego odcinka się przecinają, jeśli znajdujemy jakieś przecięcie to wstawiamy je do struktury zdarzeń Q, SortedSet zapewni aby nie znalazło się w środku zdarzenia już wcześniej wykryte.
- zdarzenie jest przecięciem odcinków **C** - zamieniamy kolejnością odcinki w strukturze stanu T, aktualizujemy położenie miotły, sprawdzamy sąsiadów odcinków tworzących przecięcie i jeśli zostało wykryte jakieś nowe przecięcie to dodajemy je do struktury zdarzeń Q

WYNIKI I ANALIZA

Do testów poddano cztery zbiory odcinków, z czego 3 były wprowadzane ręcznie, aby uwzględnić określone przypadki. Na Rys. 2 poniżej jest ich wizualizacja, zbiór 0 był zbiorem generowanym z użyciem funkcji losującej odcinki.



Rys. 2 Testowe zbiory odcinków

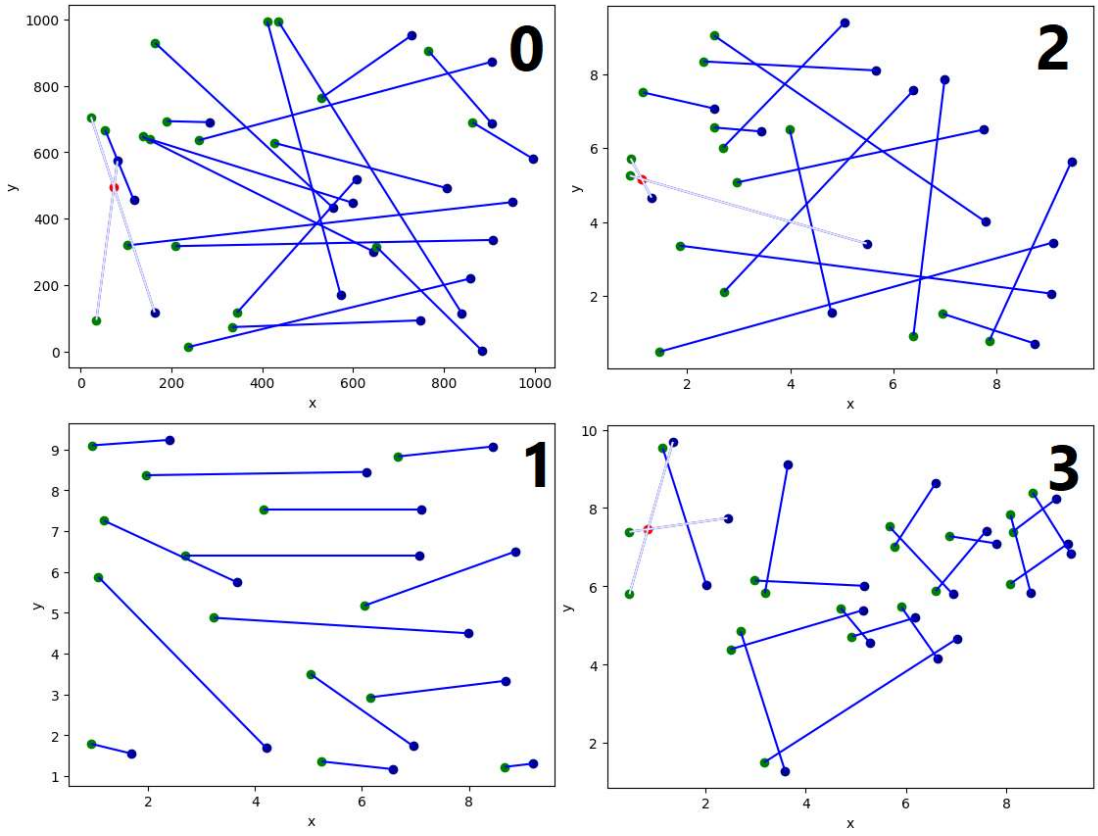
Celowo, aby zweryfikować poprawność algorytmu wybrano zbiór 1, który nie zawiera żadnych przecięć, a także zbiór 2, w którym znajduje się wiele przecięć, gdzie niektóre odcinki mają ponad 4 przecięcia. Zbiór 3 zawiera odcinki, z których każdy ma przynajmniej jedno przecięcie.

Do wizualizacji użyto następujących oznaczeń:

- **Zielone** punkty – początki odcinków
- **Granatowe** punkty – końce odcinków
- **Czerwone** punkty – punkty przecięcia się odcinków
- **Niebieski** kolor odcinka – odcinki, które aktualnie nie znajdują się w strukturze stanu (są sprocesowane, albo miotła jeszcze do nich nie dotarła)
- **Jasnoniebieski** kolor odcinka – odcinki aktualnie procesowane, znajdujące się w strukturze stanu.
- **Czarna** prosta prostopadła do osi OX - miotła

STWIERDZENIE ISTNIENIA PRZECIĘCIA

Pierwszy algorytm ma za zadanie sprawdzić czy w ogóle istnieje przecięcie w danym zbiorze punktów. Wyniki tego algorytmu są następujące, a ich wizualizację można zobaczyć na Rys. 3.



Rys. 3 Wyniki pierwszego algorytmu

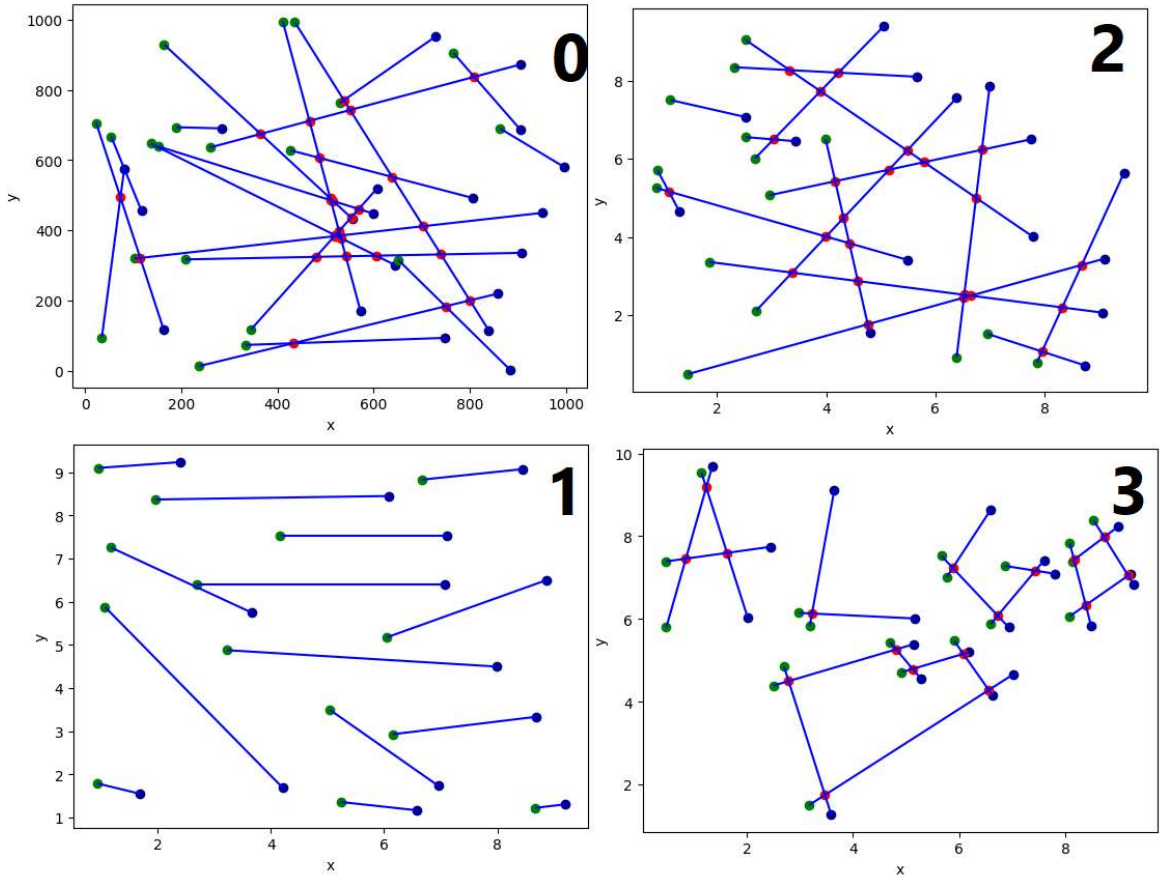
Algorytm poprawnie wyznaczył przecięcia dla zbiorów 0,2 i 3, dla zbioru 1 nie wyznaczył żadnego przecięcia zgodnie z tym jak powinno być.

WYZNACZENIE WSZYSTKICH PRZECIĘĆ

Zadaniem drugiego algorytmu było wyznaczenie wszystkich przecięć w zbiorze. Wyniki dla poszczególnych zbiorów wyglądają następująco:

- Zbiór 0 – 28 przecięć
- Zbiór 1 – 0 przecięć
- Zbiór 2 – 23 przecięcia
- Zbiór 3 – 17 przecięć

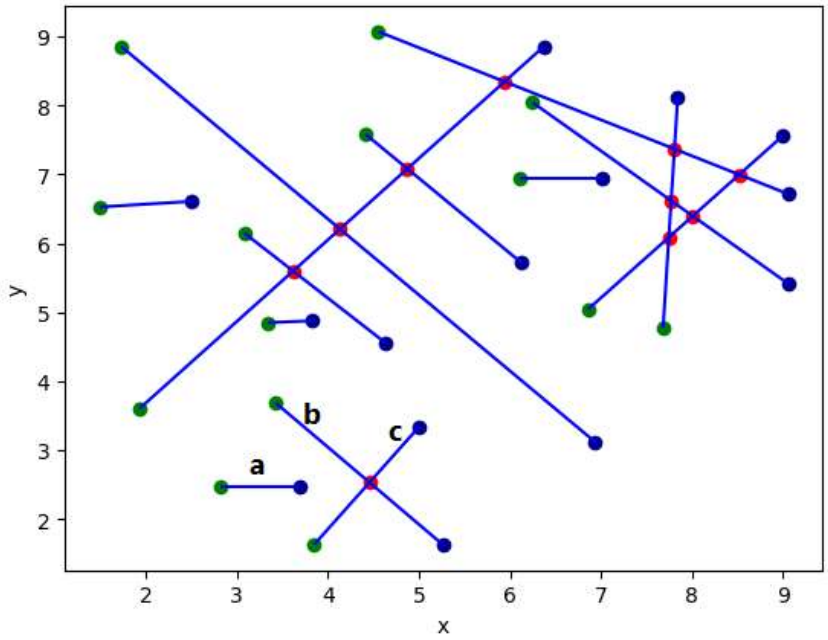
Wyniki w postaci ilustracji widnieją na Rys 4. poniżej.



Rys. 4 Wyniki drugiego algorytmu

ZBIÓR TESTUJĄCY DUPLIKATY

Istnieje opcja, aby niektóre przecięcia były wyznaczane kilkukrotnie, algorytm zapewnia, aby przecięcia w zbiorze wynikowym były zawarte bez duplikatów, do tego celu wyznaczono zbiór testowy, który widać na Rys. 5. Jest on podobny do zbioru 0 lecz ma mniejszą ilość przecięć dzięki czemu łatwo jest zweryfikować wynik. Przykładowo wstawienie duplikatu mogłoby nastąpić, gdy przecięcie zostaje wykryte w momencie usunięcia końca odcinka **a** ze struktury zdarzeń, a także w chwili dodania początku odcinka **b** do struktury zdarzeń.



Rys. 5 Wynik algorytmu na zbiorze testowym

Algorytm wyznaczył 10 przecięć dla zbioru testowego, co jest zgodne z prawdą i potwierdza poprawność tego, że algorytm nie wyznacza duplikatów.

WNIOSKI

Algorytm przeszukiwana zbioru odcinków w celu znalezienia wszystkich przecięć odcinków ma złożoność $O((P+n)\ logn)$ gdzie n to liczba odcinków w zbiorze, a P to liczba przecięć. Ma on taką złożoność dzięki strukturom bazujących na działaniach pobierania i usuwania elementu ze zbioru w czasie logarytmicznym. W tym przypadku jest to struktura działająca na posortowanym zbiorze SortedSet.