Aleksander Jóźwik gr. 2, Pon. godz. 15:00 A

Data wykonania: 19.11.2024 Data oddania: 06.12.2024

Algorytmy Geometryczne - laboratorium 4

Przecinanie odcinków

1. Dane techniczne

- System operacyjny: Fedora Linux (x86-64)
- Procesor: Intel Core i5-8350U (1.70 - 3.60 GHz)
- Pamięć RAM: 16GB (2133 MHz)
- Środowisko: Jupyter Notebook
- Język: Python 3.9.20

W realizacji ćwiczenia wykorzystano biblioteki *numpy*, *pandas*, *matplotlib*, *sortedcontainers*, *enum*, *heapq* oraz narzędzie wizualizacji stworzone przez koło naukowe *BIT*. Użyta precyzja przechowywania zmiennych i obliczeń w ćwiczeniu to *float64*.

2. Cel ćwiczenia

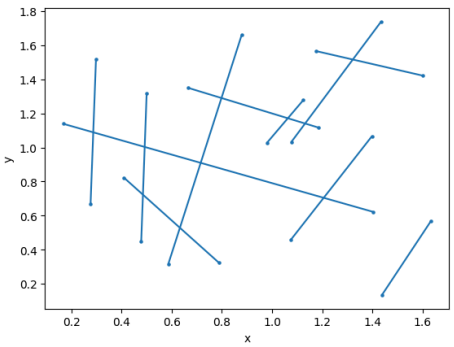
Zapoznanie się z algorytmem zamiatania w celu wykrywania przecięć odcinków w zadanym zbiorze, zarówno pod kątem stwierdzenia istnienia dowolnego przecięcia, jak i znalezienia wszystkich takich punktów. Należało także zwizualizować poszczególne kroki działania algorytmu oraz dokonać analizy porównawczej różnych struktur przechowywania danych.

3. Wstęp teoretyczny

3.1. Zagadnienia do rozpatrzenia

Dla zadanego zbioru odcinków $S = \{s_1, s_2, ..., s_n\}$ w $\mathbb{R}^2$ (przykład na Rysunku 1) należy:

- sprawdzić czy istnieje para (s_i, s_j) taka, że $i \neq j$ oraz $s_i \cap s_j \neq \emptyset$,
- znaleźć wszystkie pary (s_i, s_j) takie, że $i \neq j$ oraz $s_i \cap s_j \neq \emptyset$, wraz z punktami przecięć.



Rysunek 1: Przykładowy zbiór odcinków S w $\mathbb{R}^2$

3.2. Założenia o zadawanych zbiorach odcinków

- Żaden z odcinków nie jest pionowy,
- Dwa odcinki przecinają się w co najwyżej jednym punkcie,
- Żadne trzy odcinki nie przecinają się w jednym punkcie,
- Żadna z par odcinków nie posiada końców o tej samej współrzędnej x.

3.3. Algorytm zamiatania

Rozpatrywany w tym ćwiczeniu algorytm zamiatania jest efektywną metodą wykrywania przecięć między odcinkami na płaszczyźnie. Jego główna idea polega na wykorzystaniu tzw. miotły - wirtualnej prostej, która przesuwa się wzdłuż wybranej osi (w tym przypadku OX) i zatrzymuje się w punktach zdarzeń, gdzie następuje aktualizacja stanu i wykrywanie potencjalnych przecięć. Algorytm wykorzystuje dwie główne struktury danych. Pierwsza to struktura stanu *(T)*, która przechowuje zbiór odcinków aktywnych, uporządkowanych względem współrzędnych y. Druga to struktura zdarzeń *(Q)*, zawierająca uporządkowane rosnąco względem współrzędnych x końce odcinków oraz punkty przecięć par odcinków, które kiedykolwiek były sąsiadami w strukturze stanu. Efektywność algorytmu wynika z faktu, że sprawdzane są tylko te pary odcinków, które faktycznie mogą się przecinać, czyli te, które w danym momencie sąsiadują ze sobą w porządku pionowym określonym przez miotłę. To podejście znacząco redukuje liczbę koniecznych porównań w porównaniu z algorytmem naiwnym, który sprawdza wszystkie możliwe pary odcinków.

4. Realizacja ćwiczenia

4.1. Szczegóły implementacyjne

4.1.1. Wykorzystane struktury danych

W algorytmie sprawdzającym istnienie przecinających się odcinków w zbiorze, jako strukturę zdarzeń Q wykorzystano listę, która jest wbudowana w języku *Python*. Ponieważ algorytm kończy działanie po znalezieniu pierwszego możliwego przecięcia, w trakcie jego wykonywania nie są dodawane nowe punkty będące przecięciami odcinków. Lista inicjalizowana jest jednorazowo na początku danymi wejściowymi. Aby uzyskać amortyzowaną złożoność $O(1)$ dla operacji pobierania punktów o najmniejszej współrzędnej x, końce odcinków zostają posortowane w porządku malejącym, a w efekcie potrzebne punkty znajdują się zawsze na końcu listy.

Jako struktury stanu T użyto `SortedSet()` z biblioteki *sortedcontainers*. Pozwala ona na efektywne dodawanie, usuwanie, sprawdzanie obecności oraz wyszukiwanie elementów w złożoności $O(\log n)$. Ponadto zapewnia unikalność obiektów w T oraz pozwala na utrzymanie porządku względem współrzędnych y, co umożliwia łatwy dostęp do „sąsiadów" każdego z odcinka.

W przypadku algorytmu wyznaczającego wszystkie przecięcia odcinków, niezbędna była zmiana struktury zdarzeń. Powodem tego jest konieczność wydajnego dodawania punktów będących przecięciami odcinków, co realizowane byłoby dla listy w czasie liniowym. W tym celu skorzystano z kopca z bilioteki *heapq*. Zapewnia on dostęp do punktu o najmniejszej współrzędnej x oraz dodawanie punktów w złożoności $O(\log n)$.

Dodatkowo w celu zapewnienia poprawnego działania operacji na strukturach, zaimplementowano klasy *Point* i *Section*, które nadpisują odpowiednie metody oraz przechowują niezbędne informacje o punktach i odcinkach. Klasa *Section* wykorzystuje zmienną statyczną x, która aktualizowana w trakcie poruszania miotły, pozwala na odpowiednie określanie porządku względem y (w przypadku $y_i = y_j$ porównywane są współczynniki nachylenia prostych, na których znajdują się odcinki).

4.1.2. Metoda sprawdzania przecięć dwóch odcinków

Odcinki zapisane jako obiekty klasy *Section* przechowują parametry prostej (w postaci parametrycznej), na której leży dany odcinek. Funkcja *check_intersections* najpierw sprawdza, czy odcinki nie są równoległe, porównując ich współczynniki kierunkowe z dokładnością do ε. W przypadku, kiedy nie są one równoległe, oblicza współrzędną x punktu ich przecięcia. Następnie wyznacza wspólny zakres współrzędnych x, biorąc maksimum z początków i minimum z końców obu odcinków.

Jeśli obliczone x mieści się w tym zakresie, to oblicza współrzędną y, podstawiając x do równania jednej z prostych. Jeżeli punkt przecięcia istnieje, to funkcja zwraca krotkę (x, y), w przeciwnym przypadku zwraca *None*.

4.1.3. Implementacja algorytmu sprawdzającego czy przynajmniej jedna para odcinków w zadanym zbiorze się przecina

Algorytm rozpoczyna od utworzenia struktur zdarzeń Q i stanu T oraz zbioru służącego do przechowywania indeksów par odcinków, które zostały już sprawdzone pod kątem przecięcia. Następnie przetwarzana jest dostarczona do algorytmu lista odcinków. Dla każdego z nich tworzone są dwa zdarzenia klasy *Point*, czyli punkt lewy i punkt prawy. Zawierają one informacje o swoim typie oraz o indeksie odcinka, z którego pochodzą. Lista Q jest sortowana malejąco według współrzędnej x, aby zapewnić złożoność *O(1)* dla operacji `pop()`. Dla każdego odcinka tworzony jest obiekt klasy *Section*, który zawiera wskazania na lewy i prawy punkt oraz indeks, pod którym będzie on umieszczony w tablicy przechowującej odcinki. Algorytm przetwarza zdarzenia w pętli, do momentu aż lista Q będzie pusta. Dla każdego zdarzenia, aktualizowana jest statyczna współrzędna x klasy *Section*, która umożliwia poprawne określanie porządku w strukturze stanu dla danej pozycji miotły. Obsługiwane są dwa następujące zdarzenia:

1. Początek odcinka - odcinek przypisany do danego zdarzenia zostaje dodany do struktury stanu T, a następnie sprawdzane są przecięcia z jego sąsiadami w T. Jeżeli znaleziono przecięcie algorytm zwraca *True*.
2. Koniec odcinka - sprawdzane są przecięcia między sąsiadami odcinka w T. Jeżeli znaleziono przecięcie algorytm zwraca *True*. W przeciwnym razie odcinek jest usuwany ze struktury stanu.

Jeśli algorytm przetworzy wszystkie zdarzenia i nie znajdzie żadnego przecięcia, to zwraca *False*.

Dodatkowa optymalizacja osiągana jest poprzez wykorzystanie zbioru już sprawdzonych par odcinków. Sprawdzenie przecinania się odcinków o indeksach $\{(i, j),\ i < j\ \}$ wykonujemy tylko wtedy, gdy dana para nie znajduje się jeszcze w zbiorze. Po pierwszym sprawdzeniu przecinania się pary odcinków, dodajemy ją do zbioru przetworzonych par.

4.1.4. Implementacja algorytmu wyznaczającego wszystkie przecięcia odcinków

Algorytm stanowi modyfikację powyższego rozwiązania. Istotną różnicą jest użycie kopca jako struktury zdarzeń, z powodu opisanego w Sekcji 4.1.1. Dodatkowo wykorzystano nowy typ zdarzenia dla punktu przecięcia. Posiada on w atrybutach indeksy obu prostych, których przecięcie stanowi. Jest on dodawany do Q w momencie wykrycia przecięcia między odcinkami. Operacja ta zastępuje zwracanie informacji o powodzeniu znalezienia przecięcia. Jeżeli w trakcie przetwarzania zdarzeń z Q natrafimy na punkt przecięcia:

- Pozostawiamy tymczasowo poprzedni stan statycznego x z klasy *Section*, w celu poprawnego określania porządku w strukturze T,
- Odcinki, których przecięcie stanowi zdarzenie są usuwane z T,
- Następuje modyfikacja statycznego x, na minimalnie większe niż pozycji punktu przecięcia,
- Odcinki są ponownie dodawane do T. Ze względu na modyfikację x są one teraz zamienione miejscami.
- Sprawdzane są przecięcia z nowymi sąsiadami w strukturze stanu. Jeśli znaleziono przecięcie, punkt przecięcia jest dodawany do kopca Q.

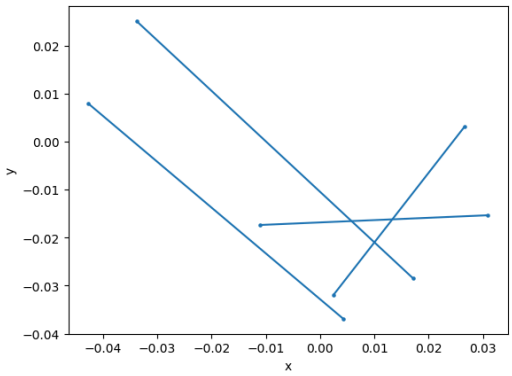
Dokładność porównań liczb zmiennoprzecinkowych oraz minimalną wartość, o którą modyfikowane jest x, wyznaczono na $\varepsilon = 10^{-12}$.

Algorytm zwraca listę krotek zawierających punkt przecięcia, wraz z oryginalnymi indeksami odcinków, do których należy. Przy pomocy pomocniczej funkcji zwracana jest także liczba wykrytych przecięć.

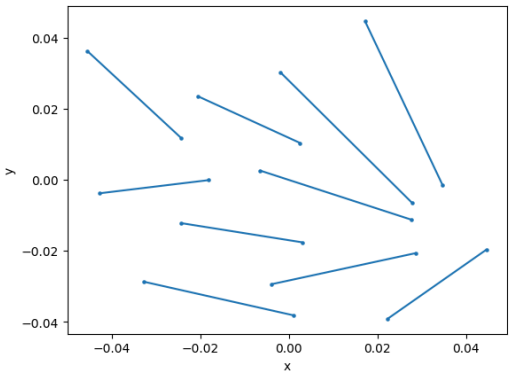
Istnieją takie układy odcinków, przy których pewne przecięcia będą wykrywane więcej niż jeden raz. Wykorzystując założenie o tym, że w jednym punkcie mogą przecinać się maksymalnie dwa odcinki oraz, że dwa odcinki przecinają się w co najwyżej jednym punkcie, problem rozwiązuje wykorzystanie zbioru przetworzonych odcinków. Gwarantuje to, że dla danej pary odcinków, sprawdzenie ich przecinania zostanie wykonane maksymalnie jeden raz. Zapobiega to także dodawaniu duplikatów punktów przecięcia do struktury zdarzeń oraz listy wynikowej. W celu zwiększenia dokładności weryfikacji, w zbiorze umieszczane są pary indeksów odcinków.

4.2. Zestawy danych testowych

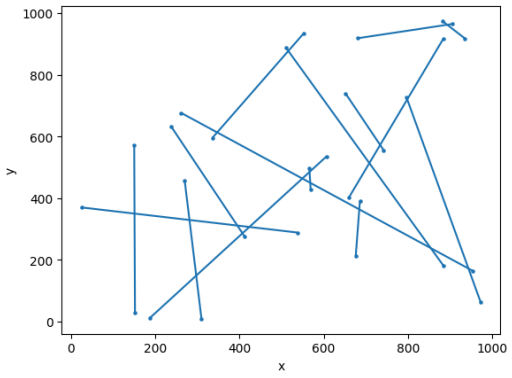
W celu przetestowania powyższych implementacji wybrano zbiory danych testowych. Odcinki były zadawane na trzy sposoby: poprzez aplikację graficzną korzystającą z biblioteki *matplotlib*, za pomocą procedury generującej losowo zadaną liczbę odcinków z podanego zakresu lub pochodziły z testów dostarczonych przez koło naukowe *BIT*.



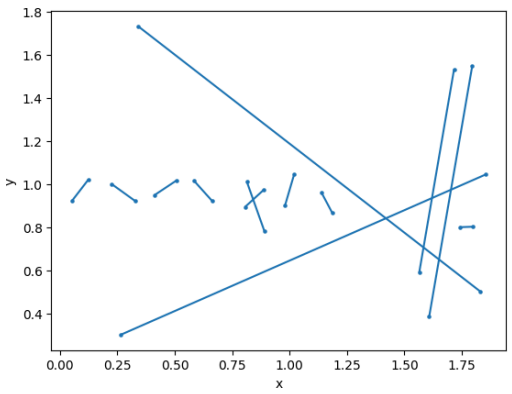
Rysunek 2: Zbiór A: zawiera 4 odcinki o niewielkiej liczbie przecięć



Rysunek 3: Zbiór B: zawiera 10 odcinków nieprzecinających się



Rysunek 4: Zbiór C: zawiera 15 losowych odcinków z zakresu $(x, y) \in [0, 1000]^2$



Rysunek 5: Zbiór D: zawiera 13 odcinków, w tym wiele krótkich następujących po sobie

5. Analiza wyników

5.1. Sprawdzanie istnienia przecięcia

Zbiór	**A**	**B**	**C**	**D**
Czy istnieje punkt przecięcia?	True	False	True	True

Tabela 1: Wyniki zwrócone przez funkcję sprawdzającą czy przynajmniej jedna para odcinków w zadanym zbiorze się przecina

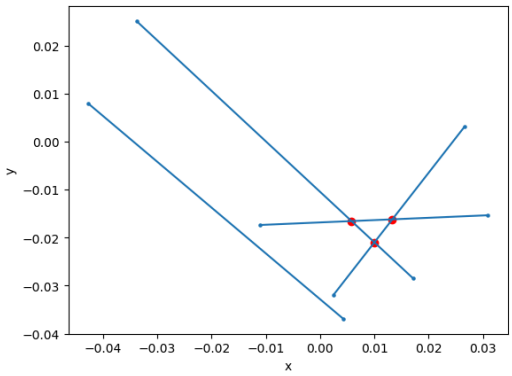
Jak można zauważyć w Tabeli 1, algorytm zwrócił poprawne wyniki dla zbiorów posiadających punkty przecięcia (Rysunki 2, 4 i 5) oraz dla zbioru ich nieposiadających (Rysunek 3).

Wizualizacje funkcjonowania algorytmu krok po kroku zostały zamieszczone w pliku *Jupyter*.

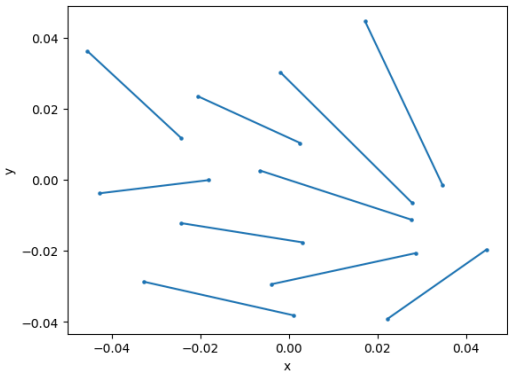
5.2. Wyznaczanie wszystkich przecięć odcinków

Poniżej znajdują się wizualizacje zwróconych przez algorytm odcinków z zaznaczonymi punktami przecięcia. Kolorem niebieskim oznaczono odcinki i ich końce, a na czerwono zaznaczono punkty przecięcia.

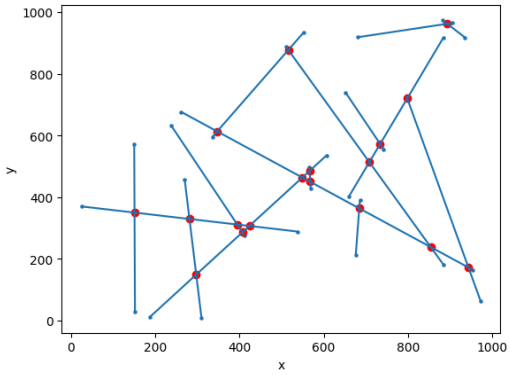
Wizualizacje algorytmów krok po kroku oraz wynikowe listy punktów przecięcia zostały wypisane w pliku *Jupyter*.



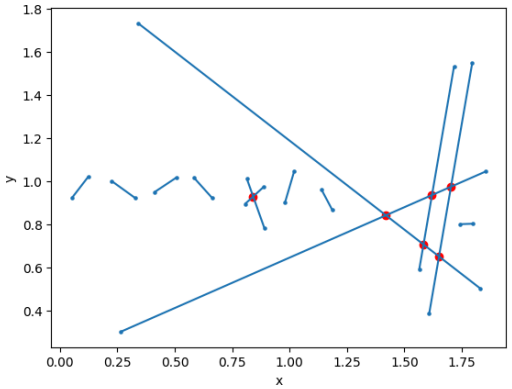
Rysunek 6: Punkty przecięcia odcinków ze zbioru A



Rysunek 7: Punkty przecięcia odcinków ze zbioru B



Rysunek 8: Punkty przecięcia odcinków ze zbioru C



Rysunek 9: Punkty przecięcia odcinków ze zbioru D

Zbiór	**A**	**B**	**C**	**D**
Liczba punktów przecięcia	3	0	18	6

Tabela 2: Liczba punktów przecięcia wyznaczonych przez algorytm

Jak można zauważyć w Tabeli 2, liczba punktów przecięcia jest zgodna z oczekiwaniami.

Algorytm zwrócił poprawne wyniki dla każdego z zadanych zbiorów danych. W przypadku zbioru A (Rysunek 6), przetestowana została obsługa struktury stanu, dla zmieniających się wraz z poruszaniem miotły wartości y odcinków. Punkty przecięcia były prawidłowo identyfikowane oraz rozpatrywane.

Weryfikacja przeprowadzona na zbiorze testowym B (Rysunek 7) wykazała prawidłowe funkcjonowanie algorytmu w scenariuszach, gdzie odcinki nie przecinają się ze sobą. W takich okolicznościach implementacja odpowiednio zarządza strukturą stanu T, wykonując operacje wstawiania oraz eliminacji odcinków dla ich końców. Potwierdza to poprawność algorytmu przy kompletnym braku występowania punktów przecięcia między analizowanymi odcinkami.

Zbiór C widoczny na Rysunku 8, posłużył do sprawdzenia jak algorytm radzi sobie z częstym przetwarzaniem punktów przecięcia oraz z wieloma operacjami na zmieniających się w czasie strukturach.

Widoczny na Rysunku 9 zbiór D posłużył do zbadania przypadku, kiedy pewne przecięcia są wykrywane wielokrotnie. Ze względu na wykorzystanie zbioru przechowującego krotki przeanalizowanych już indeksów par odcinków, każda z nich jest analizowana maksymalnie jeden raz. Wyklucza to możliwość wielokrotnego dodania tego samego punktu przecięcia do struktury zdarzeń oraz wystąpienia duplikatów w liście wynikowej. Zwiększa to także efektywność samego algorytmu.

6. Zadanie dodatkowe

Algorytm znajdujący punkty przecięcia odcinków zrealizowano tym razem przy użyciu wbudowanej w języku *Python* listy. Posłużyła ona za strukturę stanu. W celu optymalizacji wyszukiwania odcinków oraz znajdowania pozycji do wstawiania nowych, skorzystano z wyszukiwania połówkowego. Nie wpłynęło to jednak na złożoność dodawania nowych elementów, które ma złożoność $O(n)$.

W efekcie algorytm ma złożoność rzędu $O((n+k)n)$, gdzie n jest liczbą odcinków, a k liczbą wykrytych przecięć. Jest to znacząca różnica w porównaniu do implementacji opartej na *SortedSet*, gdzie złożoność wynosiła $O((n+k)\log n)$.

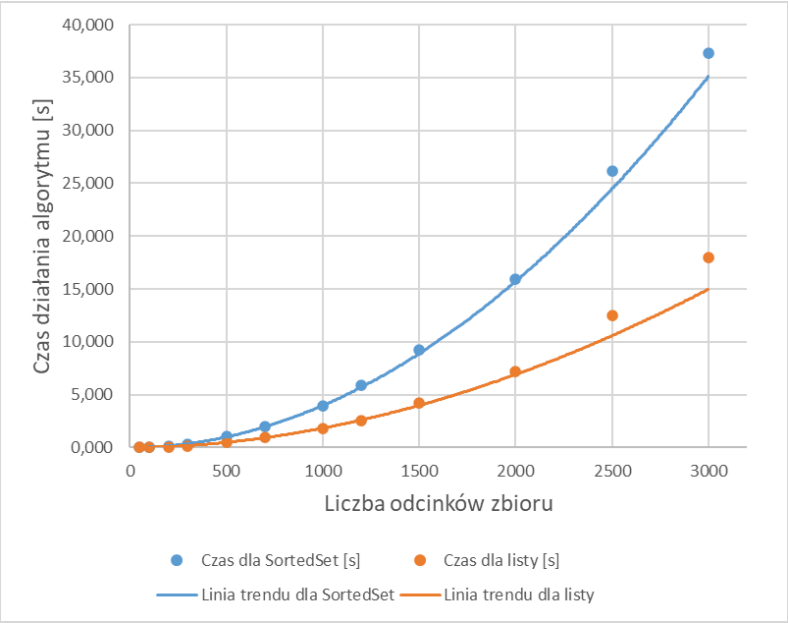
W celu przetestowania efektywności algorytmu dla obu struktur, przygotowano zbiory losowych odcinków z zakresu $(x, y) \in [0, 1000]^2$ dla zadanych kolejno liczności odcinków:

- [50, 100, 200, 300, 500, 700, 1000, 1200, 1500, 2000, 2500, 3000]

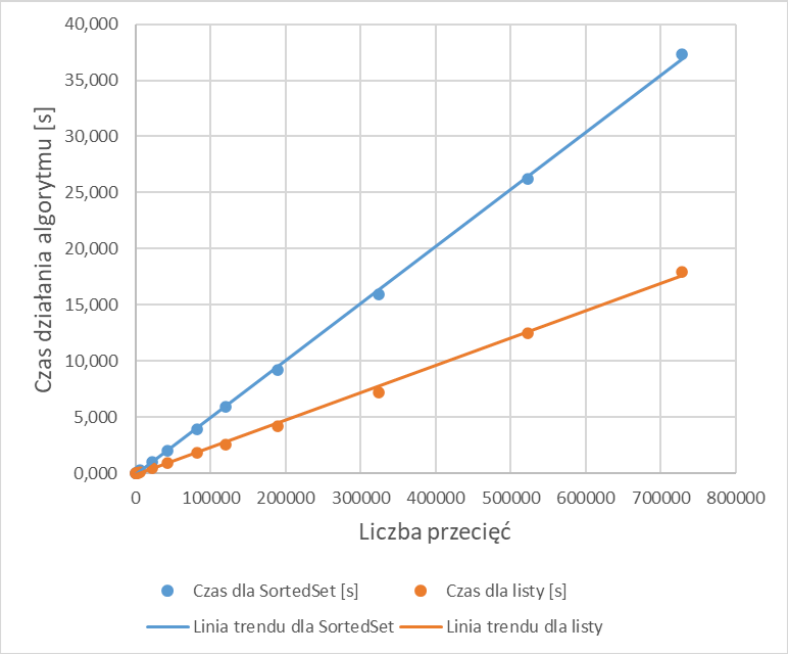
Wyniki pomiarów zamieszczono w tabeli poniżej.

Liczność zbioru		**Czas działania** [s]	
Odcinki	**Punkty przecięcia**	**SortedSet**	**Lista**
50	172	0.017	0.015
100	735	0.029	0.013
200	3800	0.148	0.061
300	6758	0.28	0.13
500	22691	1.04	0.47
700	42883	2.00	0.95
1000	81947	3.92	1.82
1200	120283	5.93	2.57
1500	189242	9.22	4.18
2000	324847	15.93	7.24
2500	522762	26.18	12.48
3000	728518	37.34	17.99

Tabela 3: Wyniki pomiaru czasu wykonania dla różnych struktur stanu



Rysunek 10: Porównanie graficzne zależności liczby odcinków zbioru od czasu wykonania algorytmu dla różnych struktur stanu



Rysunek 11: Porównanie graficzne zależności liczby przecięć odcinków od czasu wykonania algorytmu dla różnych struktur stanu

Wyniki zaprezentowane w Tabeli 3 ujawniają rozbieżność między teorią a praktyką. Wbrew oczekiwaniom wynikającym z lepszej złożoności teoretycznej implementacji wykorzystującej *SortedSet*, to właśnie wariant oparty na liście wykazał znacząco lepszą wydajność. Różnice w czasie wykonania są wyraźne zarówno w kontekście liczby przetwarzanych odcinków (Rysunek 10), jak i występujących między nimi przecięć (Rysunek 11). Może wynikać to ze specyfiki języka *Python*, w którym proste, wbudowane struktury są często zaimplementowane w wydajniejszych językach (np. C). W efekcie, bardziej zaawansowane struktury, pomimo lepszej złożoności teoretycznej, mogą okazywać się znacznie wolniejsze.

Można przypuszczać, że potencjał implementacji wykorzystującej *SortedSet* ujawniłby się w innych warunkach testowych, na przykład przy znacznie większej liczbie przetwarzanych odcinków lub w przypadkach, gdy liczba przecięć nie przekracza liczby samych odcinków.

7. Wnioski

1. Algorytm zamiatania skutecznie identyfikuje istnienie przecięć między odcinkami, jak i wyznacza wszystkie takie punkty. Wyniki uzyskane w ćwiczeniu są zgodne z oczekiwaniami i poprawnie odwzorowują charakterystykę analizowanych zbiorów danych.
2. Wybór odpowiedniej struktury danych ma kluczowe znaczenie dla działania algorytmu. Użycie *SortedSet* pozwala efektywnie zarządzać aktywnymi odcinkami w czasie zamiatania, a zastosowanie kopca ułatwia dynamiczne dodawanie nowych punktów zdarzeń.
3. Algorytm poprawnie zarządza złożonymi sytuacjami, jak wielokrotne wykrywanie tych samych odcinków czy dynamiczna aktualizacja struktury stanu. Dzięki temu unika błędów związanych z duplikacją wyników i zwiększa ogólną niezawodność.
4. Testy na różnych zbiorach danych wykazały, że złożoność obliczeniowa i czas działania algorytmu są silnie zależne od liczby przecięć w analizowanym zbiorze odcinków.
5. Algorytm wykorzystujący *SortedSet* zapewnia optymalną złożoność teoretyczną $O((n+k)\log n)$, ale w praktyce rozwiązanie oparte na prostszej strukturze (lista) okazało się wydajniejsze dla testowanych przypadków. Różnice wynikają prawdopodobnie z implementacji w języku *Python*, gdzie wbudowane struktury są zoptymalizowane na niskim poziomie.

Podsumowując, algorytm zamiatania jest efektywnym narzędziem do rozwiązywania problemu wykrywania i wyznaczania przecięć odcinków. Dzięki zastosowanym technikom i strukturom danych, takich jak *SortedSet* czy kopiec, algorytm charakteryzuje się dużą elastycznością w zarządzaniu zdarzeniami i stanem odcinków.