Dariusz Marecik gr. 4, Pon. godz. 15:00 A

Data wykonania: 18.11.2024 Data oddania: 06.12.2024

Algorytmy Geometryczne - laboratorium 4

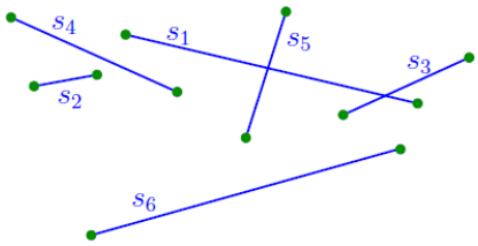
Przecinanie się odcinków

1. Cel ćwiczenia

Celem tego ćwiczenia jest poznanie algorytmu zamiatania służącego do wykrywania przecięć odcinków na płaszczyźnie oraz jego implementacja w dwóch wersjach: sprawdzenie, czy istnieje jakakolwiek para przecinających się odcinków, oraz wyznaczenie wszystkich punktów przecięć. Dodatkowo należy stworzyć wizualizację uzyskanych wyników.

2. Wstęp teoretyczny

2.1. Definicja problemu



Dany jest zbiór odcinków $S = \{s_1, s_2, ..., s_n\}$ w $\mathbb{R}^2$ (Rysunek 1)

Zadanie polega na znalezieniu wszystkich par odcinków (s_i, s_j) takich, $i \neq j$ oraz $s_i \cap s_j \neq \emptyset$ wraz z punktami przecięć.

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

2.2. Założenia odnośnie zbioru odcinków

Dla uproszczenia zagadnienia przyjąłem następujące założenia:

- Żadna para odcinków nie ma końców odcinków o tej samej współrzędnej x,
- Odcinki pionowe nie występują w zbiorze S,
- W jednym punkcie przecinają się maksymalnie dwa odcinki.

2.3. Algorytm zamiatania

Metoda zamiatania to technika algorytmiczna polegająca na przesuwaniu wirtualnej linii (zwykle poziomej lub pionowej) przez przestrzeń danych, aby stopniowo rozwiązywać problemy geometryczne lub związane z porządkowaniem, przetwarzając zdarzenia w określonej kolejności.

W naszym przypadku wirtualna miotła przemieszcza się wzdłuż osi OX, zatrzymując się w punktach należących do uporządkowanej struktury zdarzeń Q. Struktura Q zawiera punkty odpowiadające istotnym zdarzeniom, takim jak początki i końce przedziałów oraz punkty przecięć odcinków. Ponadto, podczas przetwarzania każdego zdarzenia korzystamy ze struktury stanu T, która przechowuje zbiór odcinków aktualnie przecinanych przez miotłę, umożliwiając efektywne śledzenie dynamicznych zmian w układzie.

3. Dane techniczne

3.1. Specyfikacja narzędzi

Do generowania obrazów przedstawiających odcinki wraz z punktami przecięcia i gifów przedstawiających krok po kroku działanie poszczególnych algorytmów została wykorzystana biblioteka *matplotlib* i *pandas* oraz narzędzie przygotowane przez *koło naukowe Bit*. Program został napisany w języku Python 3.10 w środowisku *Jupiter Notebook*. Wykorzystany procesor do zebrania danych to Intel Core i5-12500H 4.5 GHz pracujący w systemie Linux Mint 21.4.

3.2. Metodologia

3.2.1. Struktury danych

W algorytmie sprawdzającym, czy w danym zbiorze odcinków istnieje choć jedna para przecinających się elementów, jako struktury danych użyłem następujących rozwiązań: strukturę zdarzeń Q zaimplementowałem przy pomocy struktury z biblioteki *heapq*, natomiast dla struktury stanu T zastosowałem *SortedSet* z biblioteki *sortedcontainers*.
Struktura *SortedSet* umożliwia wydajne operacje takie jak dodawanie, usuwanie oraz sprawdzanie przynależności elementów w czasie O($\log n$), jednocześnie eliminując możliwość duplikatów. Z kolei *heapq* jest w tym przypadku w pełni wystarczające, gdyż nie wymagam modyfikacji elementów w Q ani kontroli unikalności, ponieważ przechowuje jedynie początki i końce przedziałów.

W algorytmie obliczającym wszystkie punkty przecięć odcinków zastosowałem te same struktury danych, co w poprzednim programie. Problem związany z wielokrotnym dodawaniem tych samych punktów do Q rozwiązałem poprzez wprowadzenie dodatkowego zbioru *Set*, który przechowuje już sprawdzone pary odcinków. Ponieważ w jednym punkcie mogą się przeciąć maksymalnie dwa odcinki, zapewnienie, że każda para odcinków jest sprawdzana tylko raz, gwarantuje brak duplikatów w Q.

Dla obu algorytmów opracowałem klasy `Section` oraz `Point`, które służą do reprezentacji i przechowywania kluczowych informacji odpowiednio o odcinkach i punktach. Te struktury umożliwiają efektywną organizację danych, wspierając zarówno przetwarzanie zdarzeń w strukturze Q, jak i zarządzanie aktualnym stanem w T. Opis poszczególnych atrybutów tych klas znajduje się w pliku *Jupyter* W klasie Point jest zmienna wspólna dla wszystkich instancji tej klasy, która przechowuje aktualną współrzędną x'ową miotły. Instancje klasy `Section` są porównywane tylko na bazie współrzędnej y'kowej przecięcia z miotłą.

3.2.2. Algorytm sprawdzający, czy w danym zbiorze odcinków istnieje choć jedno przecięcie

Algorytm rozpoczyna od utworzenia instancji klasy `Section` dla każdego punktu z wejściowego zbioru danych. Każdemu obiektowi `Point` przypisywana jest odpowiednia instancja klasy `Section`, co umożliwia efektywne powiązanie punktu z odcinkiem, do którego należy. Następnie wszystkie punkty są wstawiane do struktury zdarzeń Q, po czym algorytm przechodzi do głównej pętli przetwarzania.

W ramach pętli głównej algorytm pobiera kolejny punkt z Q, aktualizuje pozycję miotły na współrzędną x tego punktu, a następnie rozpatruje go w zależności od jego roli:

- Początek odcinka: Odcinek przypisany do danego punktu zostaje dodany do struktury stanu T. Następnie algorytm sprawdza, czy nowo dodany odcinek przecina się z którymkolwiek z jego sąsiadów w T. Jeśli zostanie wykryte przecięcie, algorytm natychmiast zwraca wartość True.
- Koniec odcinka: Algorytm identyfikuje sąsiadujące odcinki w T i bada, czy między nimi występuje przecięcie. Po zakończeniu tego sprawdzenia odcinek przypisany do aktualnego punktu jest usuwany ze struktury T.

Jeśli po przetworzeniu wszystkich punktów w Q algorytm nie wykryje żadnego przecięcia, zwraca wartość False.

3.2.3. Algorytm obliczający wszystkie punkty przecięć odcinków

Algorytm stanowi rozszerzenie poprzedniego podejścia. Podczas gdy poprzednia wersja algorytmu kończyła działanie, gdy wykryto pierwsze przecięcie (zwracając wartość True), niniejszy algorytm kontynuuje pracę, zapisując punkt przecięcia do struktury zdarzeń Q wraz z odpowiednią etykietą oraz parą odcinków, które w nim się przecinają. Przed dodaniem takiego punktu algorytm weryfikuje, czy znajduje się on po prawej stronie obecnej pozycji miotły, aby zapewnić poprawność kolejności przetwarzania.

Ponadto wprowadza się dodatkową kategorię punktów — punkt przecięcia. Podczas przetwarzania takiego zdarzenia algorytm wykonuje następujące kroki:

- Pobiera odcinki, które przecinają się w tym punkcie, a następnie tymczasowo przesuwa pozycję miotły minimalnie w lewo i usuwa te odcinki ze struktury T.
- Miotła jest przesuwana na współrzędną minimalnie większą od współrzędnej punktu przecięcia, po czym usunięte odcinki są ponownie dodawane do T. W efekcie ich kolejność w strukturze T zostaje zaktualizowana, odzwierciedlając zamianę miejsc wynikającą z przecięcia.
- Następnie algorytm sprawdza, czy nowo uporządkowane odcinki przecinają się z ich nowymi sąsiadami w T, i odpowiednio aktualizuje strukturę zdarzeń Q, dodając nowe punkty przecięcia, jeśli zostaną wykryte. Dokonywane są dwie operacja sprawdzania par odcinków. Jedna dla odcinka, który jest wyżej pośród nowo dodanych i jego sąsiada powyżej, a druga dla pozostałego odcinka nowo dodanego i jego najbliższego niższego sąsiada.

Taka konstrukcja zapewnia poprawne śledzenie zmian konfiguracji odcinków oraz umożliwia wyznaczenie wszystkich punktów przecięć w zadanym zbiorze. Minimalne przesunięcie miotły ε zostało wyznaczone eksperymentalnie i wynosi 10^{-8}. Wartość ta została dobrana w taki sposób, aby zapewnić odpowiednią precyzję w obliczeniach przy jednoczesnym uniknięciu problemów związanych z błędami numerycznymi, które mogłyby wystąpić przy zbyt małych wartościach przesunięcia.

Algorytm zwraca krotki, w których znajdują się współrzędne punktu przecięcia odcinka oraz indeksy odcinków, które w tym punkcie się przecinają,

Dodatkowo jest poczyniona optymalizacja w postaci zbioru już sprawdzonych ze sobą odcinków. Dzięki niemu nie są rozpatrywane kilka razy te same odcinki, co zapobiega duplikowaniu punktów w zwracanej tablicy przecięć.

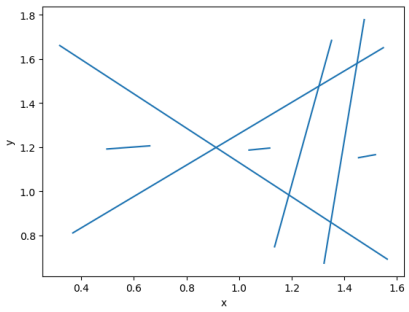
Przesunięcie miotły o ε w lewo jest niezbędne, ponieważ dwa odcinki posiadające tę samą współrzędną y są traktowane jako równoważne przez funkcję porównującą te odcinki w strukturze *SortedSet*. W konsekwencji, nie jest spełniony wymóg unikalności elementów w strukturze *SortedSet*, co prowadzi do jej nieprawidłowego działania. Eksperymenty wykazały, że ustawienie miotły dokładnie na współrzędnej x rozpatrywanego punktu skutkuje błędami – odcinki, które powinny być w *SortedSet*, nie są w nim znajdowane.

Przeprowadziłem także testy zmodyfikowanej funkcji porównującej, która dla odcinków o tych samych współrzędnych y przecięcia z miotłą dodatkowo porównywała ich współczynniki kierunkowe. Chociaż to rozwiązanie było teoretycznie poprawne, okazało się obarczone znacznym błędem numerycznym. W przypadku dużych zbiorów, struktura *SortedSet* miała trudności z prawidłową lokalizacją odpowiednich odcinków w swojej strukturze danych. Problem ten występował niezależnie od precyzji zera przy porównywaniu równości współrzędnych y.
Natomiast technika polegająca na cofnięciu miotły o ε nie napotkała takich trudności.

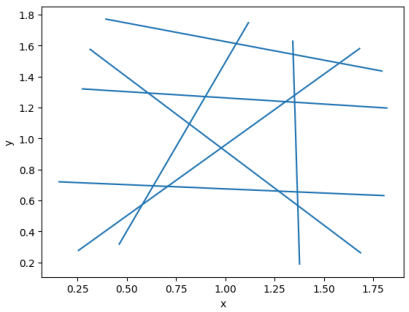
4. Program ćwiczeń

Na początku przystąpiłem do stworzenia narzędzia graficznego pozwalającego wprowadzać w sposób interaktywny kolejne odcinki. Następnie za pomocą tego narzędzia stworzyłem 4 układy testujące dla algorytmu zamiatania. Są nimi:

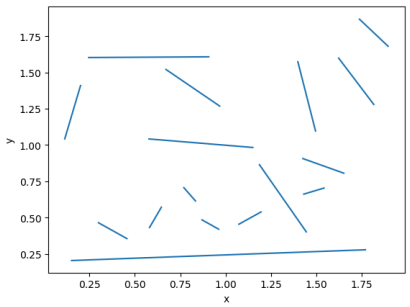
- **Zbiór A** - posiada 7 odcinków z małą liczbę przecięć (Rysunek 2)
- **Zbiór B** - posiada 7 odcinków z gęstą sieć przecięć (Rysunek 3)
- **Zbiór C** - posiada 16 odcinków bez przecięć (Rysunek 4)
- **Zbiór D** - posiada 14 odcinków, w tym wiele bardzo krótkich następujących po sobie (Rysunek 5)



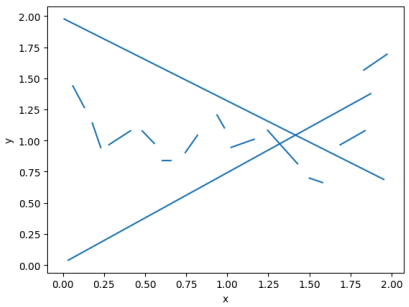
Rysunek 2: Zbiór odcinków A



Rysunek 3: Zbiór odcinków B



Rysunek 4: Zbiór odcinków C



Rysunek 5: Zbiór odcinków D

Przygotowane zbiory testowe zostały dobrane w sposób umożliwiający wszechstronne sprawdzenie działania algorytmu w różnych scenariuszach:

Układ odcinków w zbiorze A pozwala na weryfikację algorytmu pod względem prawidłowej obsługi punktów przecięcia oraz poprawnego sprawdzania odpowiednich par odcinków w trakcie obsługi tych punktów.
Zbiór B zawiera wiele punktów przecięć, co umożliwia ocenę efektywności algorytmu w obsłudze zdarzeń związanych z wykrywaniem i przetwarzaniem przecięć odcinków.
Zbiór C nie posiada punktów przecięcia. Testuje algorytm pod kątem operacji wykonywanych na punkach początku i końca odcinka oraz sprawdza poprawne nieznajdowanie przecięcia dla dwóch odcinków niespełniających warunków.
Zbiór D charakteryzuje się szczególną trudnością wynikającą z częstego rozpoczynania i kończenia odcinków w sytuacjach, gdy struktura stanu T zawiera jedynie dwa odcinki. Ten scenariusz sprawdza poprawność algorytmu w sytuacji potencjalnego wielokrotnego dodawania tego samego punktu do struktury Q.

5. Analiza wyników

5.1. Algorytm sprawdzający, czy w danym zbiorze odcinków istnieje choć jedna para przecinających się elementów

Zwrócone wartości przez program dla danych przypadków zostały wpisane do tabeli:

Testowany zbiór odcinków	Zbiór A	Zbiór B	Zbiór C	Zbiór D
Czy występuje przecięcie?	True	True	False	True

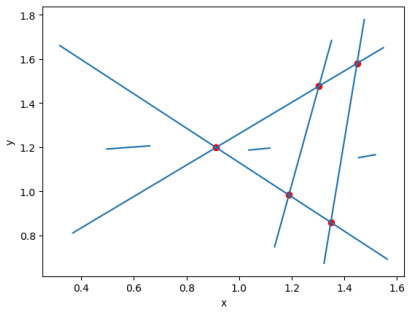
Tabela 1: Wyniki działania algorytmu dla zbiorów testujących

Tablica Tabela 1 przedstawia skuteczność działania algorytmu w odniesieniu do różnych zbiorów testowych. Wyniki wskazują, że dla przypadków A, B oraz D algorytm poprawnie zidentyfikował pierwszy punkt przecięcia odcinków i zwrócił oczekiwaną wartość. W przypadku zbioru C, algorytm zwrócił wartość False, co oznacza brak wykrycia punktów przecięcia w zadanym zbiorze odcinków. Wynik ten jest zgodny ze stanem faktycznym, co potwierdza poprawność algorytmu również w przypadku braku kolizji.

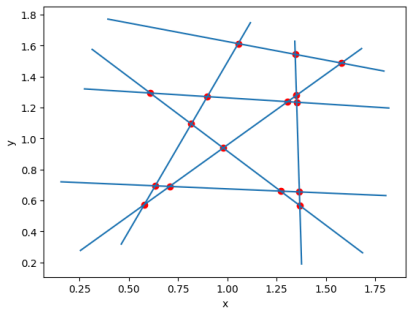
5.2. Algorytm obliczający wszystkie punkty przecięć odcinków

Na poniższych rysunkach zamieszczam wizualizacje zwróconych przez algorytm punktów przecięcia dla zbiorów testowych.
Legenda obrazów:

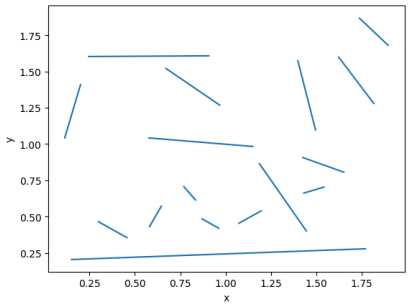
- *kolor niebieski* - odcinki zawarte w danym zbiorze,
- *kolor czerwony* - punkty przecięcia odcinków.



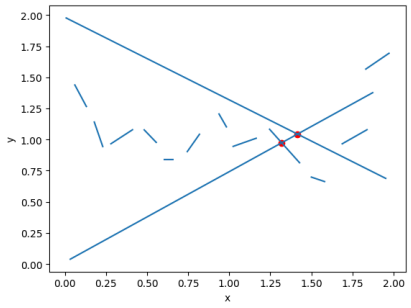
Rysunek 6: Punkty przecięcia w zbiorze A



Rysunek 7: Punkty przecięcia w zbiorze B



Rysunek 8: Punkty przecięcia w zbiorze C



Rysunek 9: Punkty przecięcia w zbiorze D

Testowany zbiór odcinków	Zbiór A	Zbiór B	Zbiór C	Zbiór D
Liczba punktów przecięcia	5	16	0	2

Tabela 2: Liczba znalezionych przecięć dla danych zbiorow

Dla badanych zbiorów liczba punktów przecięcia zawarta w Tabela 2 zgadza się z oczekiwaniami. Podobnie jest ze zwróconymi pozycjami punktów przecięcia pokazanych na Rysunek 6, Rysunek 7, Rysunek 8 i Rysunek 9.

Zbiory testowe A i B wykazały, że algorytm poprawnie obsługuje wszystkie kategorie punktów zdarzeń. W przypadku zbioru A algorytm skutecznie zarządzał strukturą stanu T i prawidłowo identyfikował punkty przecięcia wraz z wybieraniem odpowiednich odcinków do sprawdzenia, czy się przecinają w trakcie obsługi przecięcia. Z kolei dla zbioru B, charakteryzującego się częstym występowaniem i przetwarzaniem punktów przecięcia w krótkich odstępach, algorytm wykazał zdolność sprawnego operowania na dynamicznie zmieniających się strukturach i poprawnie wyznaczył interesujące nas punkty przecięcia.

Zbiór C potwierdził, że algorytm poprawnie obsługuje przypadki, w których brak jest punktów przecięcia. W takiej sytuacji algorytm prawidłowo zarządza strukturą stanu T, skutecznie dodając i usuwając odcinki w odpowiednich momentach, co zapewnia poprawność działania nawet w przypadku całkowitego braku kolizji między odcinkami.

Zbiór D został zaprojektowany w celu oceny działania algorytmu w scenariuszach, gdzie może wystąpić wielokrotne dodawanie tych samych odcinków do struktury stanu T. Taka sytuacja może mieć miejsce, gdy między dwoma odcinkami przecinającymi się w przyszłym punkcie przecięcia znajdują się odcinki, których początki i końce są przetwarzane sekwencyjnie. W efekcie, podczas obsługi punktu końcowego jednego z takich odcinków, algorytm mógłby ponownie sprawdzać przecięcie tych samych dwóch bazowych odcinków. Dzięki zastosowaniu struktury *Set* dla sprawdzonych już odcinków algorytm zwrócił prawidłową liczbę punktów przecięcia, zgodną z oczekiwaniami teoretycznymi. Wyniki te zostały przedstawione w tabeli Tabela 2, co potwierdza skuteczność mechanizmu eliminacji zduplikowanych operacji oraz poprawność implementacji algorytmu w testowanych scenariuszach.

6. Wizualizacje działania algorytmów

Wszystkie wizualizacja działania algorytmów dla powyższych zbiorów zostały zawarte w pliku *Jupyter*.

Legenda kolorów obiektów prezentowanych na gif'ach:

- ***kolor niebieski*** - odcinki wraz z punktami początkowymi i końcowymi nieznajdujące się jeszcze w strukturze stanu
- ***kolor zielony*** - odcinki wraz z punktami początkowymi i końcowymi, które znajdują się aktualnie w strukturze stanu T
- ***kolor czarny*** - odcinki, które zostały usunięte z struktury stanu T i punkty, który zostały już przetworzone
- ***kolor żółty*** - odcinki, których sprawdzamy przecięcie
- ***kolor czerwony*** - znalezione punkty przecięć odcinków

7. Zadanie dodatkowe

7.1. Metodologia

W algorytmie wyznaczającym wszystkie punkty przecięcia odcinków dokonano zmiany struktury stanu T z *SortedSet* na zwykłą listę wbudowaną w język Python. Dodawanie nowego odcinka do tej struktury polega na znalezieniu odpowiedniego indeksu za pomocą wyszukiwania binarnego (bsearch) i umieszczeniu elementu w odpowiedniej pozycji. Operacja ta jest jednak obliczeniowo kosztowna, ponieważ w najgorszym przypadku złożoność wyszukiwania binarnego wynosi O($\log n$), a umieszczenie elementu wiąże się z przesuwaniem innych elementów w strukturze, co prowadzi do łącznej złożoności O(n) na operację dodawania.

Z drugiej strony, operacja zamiany miejscami odcinków w tej strukturze podczas rozpatrywania punktu przecięcia jest szybsza w przypadku listy niż w przypadku *SortedSet*. Dla listy wystarczy zamienić miejscami dwa elementy, co odbywa się w czasie stałym O(1). Natomiast w przypadku *SortedSet* operacja ta wymaga wyjęcia elementów z struktury, a następnie ponownego wstawienia, co wiąże się z większymi kosztami obliczeniowymi, głównie z powodu konieczności utrzymania zrównoważonej struktury.

7.2. Porównanie złożoności czasowej

Operacje dodawania, usuwania oraz wyszukiwania elementów w liście mają złożoność czasową O(n), co skutkuje całkowitą złożonością algorytmu na poziomie O($(n + k)n$), gdzie k oznacza liczbę wykrytych przecięć. W pesymistycznym przypadku, gdy liczba przecięć jest rzędu O(n^2), złożoność może wzrosnąć do O(n^3).

W porównaniu do wcześniejszej implementacji, gdzie struktura stanu opierała się na *SortedSet*, złożoność algorytmu w najlepszym przypadku wynosiła O($(n + k) \log n$), a w pesymistycznym przypadku O($n^2 \log n$). Przejście na listę powoduje zatem znaczący wzrost złożoności obliczeniowej, szczególnie w przypadku dużych zbiorów danych.

7.3. Zbiory testujące

Zostały wygenerowane dwa zbiory testowe. Jeden został stworzony przy użyciu zaimplementowanej procedury, która umożliwia losowe rozmieszczanie odcinków o stałej długości na płaszczyźnie przy zadanej liczbie odcinków, drugi zaś przy pomocy metody, która nie zakłada równości długości odcinków. Dzięki temu uzyskano zbiory testowe, które w jednym przypadku charakteryzują się względnie stałą liczbą przecięć między odcinkami, a w drugim liczba przecięć rośnie kwadratowo wraz ze wzrostem n. Dokładna specyfika danych zbiorów została przedstawiona w tabelach poniżej.

Pierwszy zbiór testowy został zaprojektowany w celu analizy algorytmów w warunkach, w których liczba odcinków n ulega zmianie, przy jednoczesnym minimalizowaniu wpływu liczby punktów przecięcia k. Z kolei drugi zbiór testowy umożliwia badanie zachowania algorytmów w sytuacji, gdy liczba punktów przecięcia k rośnie wraz ze wzrostem n, co pozwala na ocenę ich wydajności w bardziej złożonych przypadkach.

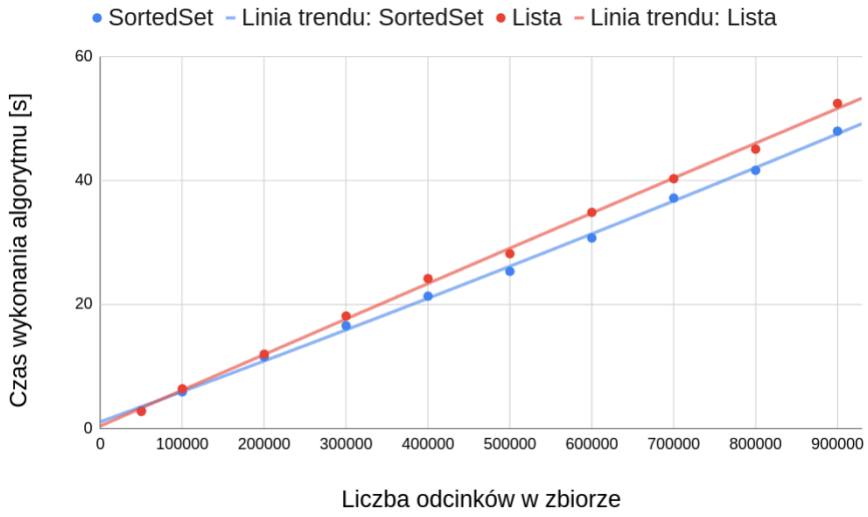
7.4. Opracowanie wyników

7.4.1. Zbiór ze stałym k

Liczebność badanych zbiorów oraz liczba punktów przecięcia w nich została umieszczona w tabeli poniżej:

Liczność zbiorów		Czas wykonania [s]	
Zbiór odcinków	Punkty przecięcia	SortedSet	List
50000	28085	2,889	2,758
100000	28365	5,918	6,409
200000	28398	11,561	11,986
300000	28756	16,582	18,134
400000	28659	21,340	24,179
500000	28725	25,343	28,182
600000	28891	30,717	34,844
700000	28692	37,154	40,280
800000	28559	41,629	45,042
900000	28523	47,944	52,417

Tabela 3: Czasy wykonania algorytmów z użyciem poszczególnych struktur dla stałego k



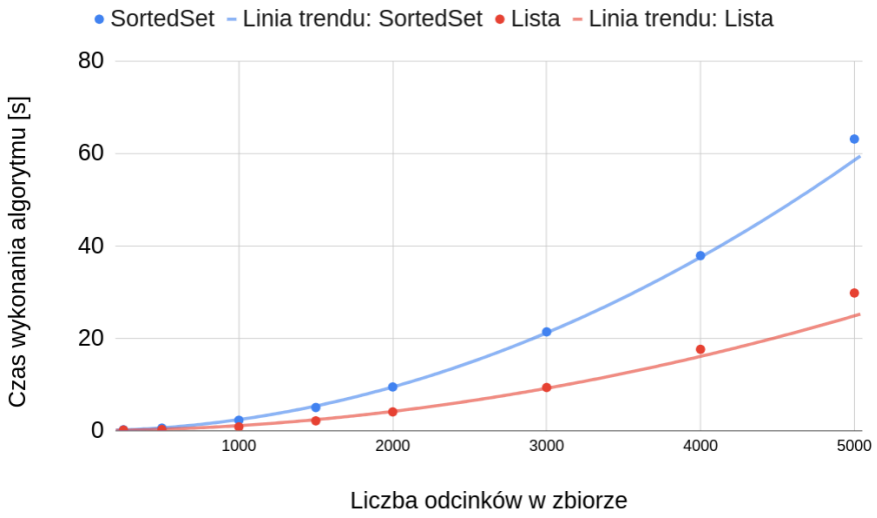
Rysunek 10: Wykres porównawczy czasu znalezienia punktów przecięcia dla różnych implementacji struktury T dla stałego k

Z danych zawartych w Tabela 3 możemy zauważyć, że wykresy złożoności czasowej dla obu algorytmów mają kształt zbliżony do liniowego, co ilustruje Rysunek 10. Tego rodzaju zachowanie wynika z specyfiki języka Python. Operacje dodawania i usuwania elementów z listy są wyjątkowo szybkie, dzięki implementacji tych metod w języku C, co skutkuje zminimalizowaniem czasu tych operacji w porównaniu do oczekiwanej teoretycznej złożoności. Dodatkowo, szybkość operacji logarytmicznych zarówno na liście (np. wyszukiwanie binarne) jak i na *SortedSet* prowadzi do zanikania różnicy między złożonościami $O(n)$ a $O(n \log n)$ w analizowanym zakresie danych.

Zauważalna jest jednak niewielka przewaga implementacji opartej na *SortedSet* nad wersją wykorzystującą listę, a ta różnica wzrasta wraz z rosnącą wartością n. Warto jednak podkreślić, że dla mniejszych wartości n algorytm oparty na liście okazał się szybszy.

7.4.2. Zbiór z k rosnącym w sposób kwadratowy

d



Rysunek 11: Wykres porównawczy czasu znalezienia punktów przecięcia dla różnych implementacji struktury T dla k rosnącego w sposób kwadratowy

Na Rysunek 11 or widać, że wykresy czasu wykonania dla obu struktur przyjmują kształt funkcji kwadratowej. Wynika to głównie z rosnącej liczby punktów przecięcia k, która wzrasta kwadratowo i odgrywa dominującą rolę w algorytmie dla obu struktur.

W tym przypadku zauważalna jest dwukrotna przewaga implementacji opartej na liście w porównaniu do *SortedSet*, co jest efektem szybszej obsługi punktów przecięcia przez listę w kontekście tego algorytmu.

8. Podsumowanie

- Algorytm sprawdzający, czy istnieje jakiekolwiek przecięcie odcinków, prawidłowo identyfikuje sytuacje kolizyjne i zwraca odpowiednie wartości dla każdego testowanego przypadku. Algorytm obliczający wszystkie punkty przecięć działa poprawnie dla różnych zbiorów odcinków, w tym tych o gęstej sieci przecięć oraz takich, gdzie nie występują kolizje.
- Wybór struktur heapq dla kolejki zdarzeń i SortedSet dla struktury stanu umożliwił wydajne przetwarzanie danych w czasie logarytmicznym. Dodanie zbioru Set dla sprawdzonych par odcinków wyeliminowało problem duplikatów punktów przecięć, co poprawiło precyzję wyników.
- Zestawy testowe (A, B, C, D) zostały dobrane tak, aby uwzględnić różnorodne przypadki, od prostych do złożonych. Analiza wyników potwierdziła, że algorytmy radzą sobie zarówno w scenariuszach z wieloma przecięciami, jak i w sytuacjach braku kolizji.
- Wprowadzenie minimalnego przesunięcia miotły $\varepsilon = 10^{-8}$ rozwiązało problemy z porównywaniem współrzędnych i umożliwiło poprawne działanie struktury *SortedSet*.
- Wyniki w pełni potwierdziły poprawność zaimplementowanych rozwiązań, zarówno pod kątem obsługi przecięć, jak i efektywności struktur danych.
- Analiza czasowa algorytmów pokazała, że dla stałej liczby przecięć lepszą opcją jest struktura zdarzań oparta na *SortedSet*, zaś dla dużej liczby przecięć w zbiorze lepsza jest implementacja z listą pythonową.