1 Wstęp

Celem ćwiczenia było zaimplementowanie algorytmu szukającego przecięć w zadanym zbiorze odcinków metodą "zamiatania".

1.1 Wstęp teoretyczny

Implementując algorytm przyjęliśmy następujące założenia:

- Wszystkie odcinki leżą na jednej płaszczyźnie.
- Żaden odcinek nie jest pionowy (równoległy do osi OY).
- Współrzędne x odcinków są parami różne.
- Odcinki przecinają się w co najwyżej jednym punkcie.
- W jednym punkcie przecinają się co najwyżej dwa odcinki.

Algorytm korzystał z metody "zamiatania", przesuwając "miotłę" (prostą równoległą do osi OY) w miejsce kolejnych zdarzeń trzymanych w strukturze Q i aktualizując strukturę stanu T, przetrzymującą odcinki obecnie przecinające się z miotłą.

1.1.1 Struktura zdarzeń Q

Zdarzeniami w algorytmie są początki, końce oraz miejsca przecięć odcinków posortowane rosnąco względem współrzędnej x. Struktura ta musi wspierać szybkie dodawanie nowych zdarzeń oraz odczyt kolejnego zdarzenia.

W mojej implementacji algorytmu strukturą zdarzeń jest drzewo czerwono-czarne, na którym operacje dodawania i odczytu wykonywane są ze złożonością obliczeniową $O(logn)$, gdzie n oznacza liczbę elementów w drzewie. Użyłem właśnie tej struktury aby uniknąć dodawania dwukrotnie tego samego przecięcia jako zdarzenia; algorytm przed dodaniem zdarzenia sprawdza, czy w strukturze zdarzeń już się ono nie znajduje (zdarzenia jesteśmy w stanie rozróżnić przechowując czy reprezentują początek, koniec lub przecięcie odcinków oraz indeksy odcinków do których się odnosi). Z tego powodu także program nigdy nie usuwa zdarzeń, jedynie przechodząc do kolejnego.

Inną stukturą, która mogła być wykorzystana jest przykładowo kolejka priorytetowa, a w przypadku algorytmu sprawdzającego tylko czy istnieje przecięcie, wystarzy użycie odpowiednio posortowanej listy (algorytm ten nie dodaje nigdy zdarzeń).

1.1.2 Struktura stanu T

Stan programu reprezentowany jest przez zbiór odcinków obecnie zamiatanych przez miotłę, posortowany względem współrzędnej y punktu przecięcia z miotłą. Struktura ta musi wspierać szybkie dodawanie, usuwanie oraz odczyt odcinków.

Podobnie jak dla Q, w mojej implementacji strukturą stanu jest drzewo czerwono-czarne. Program przechowuje w niej indeksy zamiatanych odcinków porównując je funkcją, która oblicza ich punkty przecięcia z miotłą. Użycie takiej funkcji porównującej sprawia, że struktura nie musi na nowo sortować wartości po dodaniu nowej. Kolejność dwóch odcinków w strukturze zmienia się jedynie w miejscu ich przecięcia, co pozwala nam je wtedy zamienić miejscami, bez zmieniania wewnętrznej struktury drzewa.

1.1.3 Algorytm zamiatania

Algorytm obsługuje kolejno zdarzenia ze struktury Q w następujący sposób:

- Jeżeli zdarzenie jest początkiem odcinka, to dodajemy go do T. Następnie sprawdzamy czy przecina się on z odcinkami sąsiadującymi w strukturze stanu.
- Jeżeli zdarzenie jest końcem odcinka, to usuwamy go z T i sprawdzamy czy jego byli sąsiedzi przecinają się ze sobą.
- Jeżeli zdarzenie jest przecięciem odcinków, to zamieniamy ich kolejność w strukturze stanu, a następnie sprawdzamy czy przecinają się z ich nowymi sąsiadami.

We wszystkich przypadkach, jeżeli znajdziemy dwa przecinające się odcinki, to dodajemy nowe zdarzenie do Q, sprawdzając czy wcześniej już go nie dodaliśmy (aby uniknąc sprawdzania wielokrotnie tego samego przecięcia). Przeciecie odcinków znajdujemy przyrównując postacie kierunkowe prostych przechodzących przez początki i końce odcinków i sprawdzając czy znaleziony punkt zawiera się w obu odcinkach.

Ostatecznie wynikiem programu jest lista trójek zawierających współrzędne przecięcia oraz indeksy przecinających się odcinków.

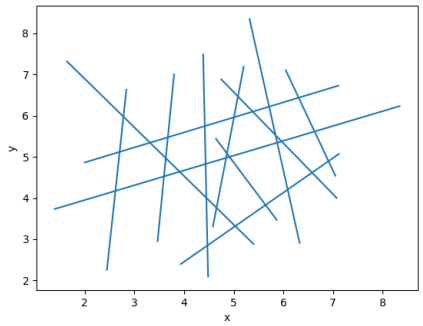
Złożoność obliczeniowa:	$O((p+n)logn)$	gdzie n oznacza liczbę odcinków,
Złożoność pamięciowa:	$O(p+n)$	a p oznacza liczbę przecięć.

1.2 Specyfikacja narzędzi i sprzętu

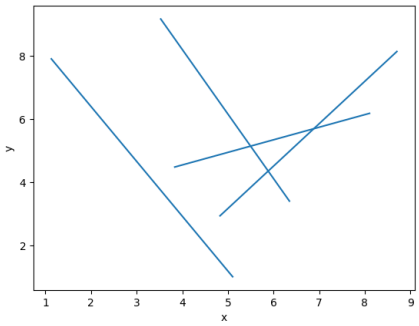
Wszystkie potrzebne obliczenia zostały wykonane przy użyciu interpretera języka Python wersji 3.12. Ponadto, w celu wygenerowania zbiorów punktów użyłem biblioteki `numpy`. Wykresy i wizualizacja wyników została przygotowana za pomocą narzędzia przygotowanego przez koło naukowe Bit. Do implementacji interaktywnego zadawania odcinków użyłem biblioteki `matplotlib`. Kod znajduje się w załączonym pliku. Przedstawione wyniki zostały wygenerowane na komputerze z systemem operacyjnym Debian 12 i procesorem AMD Ryzen 5 3600 3.6GHz.

2 Realizacja obliczeń

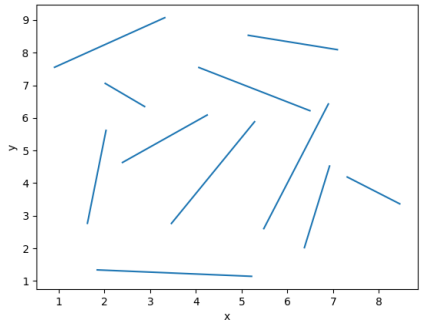
Realizację ćwiczenia zacząłem od implementacji struktury drzewa czerwono-czarnego oraz implementacji dwóch wersji omawianego algorytmu: `is_intersection`, sprawdzającą czy istnieje jakiekolwiek przecięcie oraz `find_intersections`, szukającą wszystkich przecięć. Następnie dodałem do programu możliwość zadawania własnych odcinków za pomocą myszki lub generacji losowych zbiorów odcinków. Ostatecznie przygotowałem cztery zbiory testowe, sprawdzające czy algorytm działa poprawnie w danych przypadkach. Zbiory te przedstawione są na Rysunku 1.



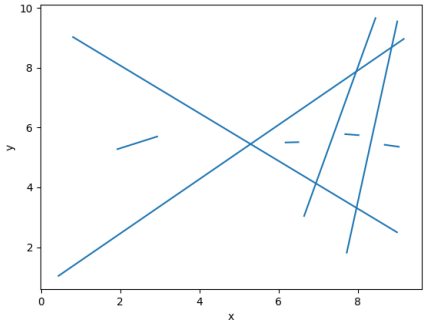
(i) Zbiór A



(ii) Zbiór B



(iii) Zbiór C



(iv) Zbiór D

Rysunek 1: Wybrane zbiory testowe

3 Wyniki dla zbiorów testowych

Sekcje od 3.1 do 3.4 zawierają ilustracje wyników działania algorytmu dla zbiorów testowych wraz z krótkim komentarzem na temat tego co testują. Algorytm `is_intersection` zwraca wartość `True` jeżeli istnieje conajmniej jedno przecięcie lub `False` w przeciwnym przypadku. Rysunki 2, 3, 4, 5 zawierają znalezione przez algorytm `find_indersections` przecięcia oznaczone kolorem czerwonym. Ponadto, razem ze sprawozdaniem zawarte są animacje pokazujące kolejne kroki działania algorytmu dla każdego zbioru. W animacjach przyjmuję następujące oznaczenia:

- Kolor niebieski - odcinki, do których miotła jeszcze nie doszła.
- Kolor zielony - odcinki w strukturze stanu.
- Kolor żółty - odcinki, między którymi algorytm obecnie szuka przeciecia.
- Kolor szary - odcinki sprawdzone, poza strukturą stanu.
- Punkty czerwone - znalezione punkty przeciecia.
- Czerwona linia - obecna pozycja miotły.

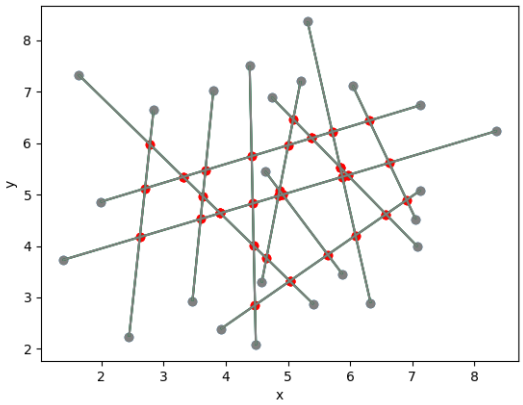
Zawarty ze sprawozdaniem kod wypisuje także znalezione przeciecia w formie tekstowej.

3.1 Zbiór A

Zbiór testuje czy algorytm znajdzie wszystkie przecięcia w przypadku gdzie wszystkie odcinki przecinają się conajmniej z trzema innymi.

Wyniki:

`is_intersectiom`: `True`



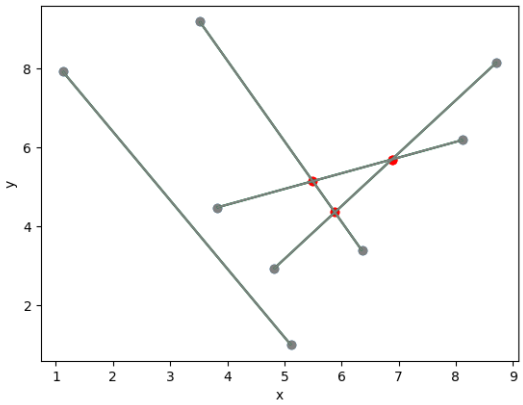
Rysunek 2: Ilustracja wyniku `find_intersections` dla zbioru A

3.2 Zbiór B

Zbiór testuje czy algorytm poprawnie działa dla małej liczby przecięć oraz czy algorytm poprawnie sortuje strukturę stanu.

Wyniki:

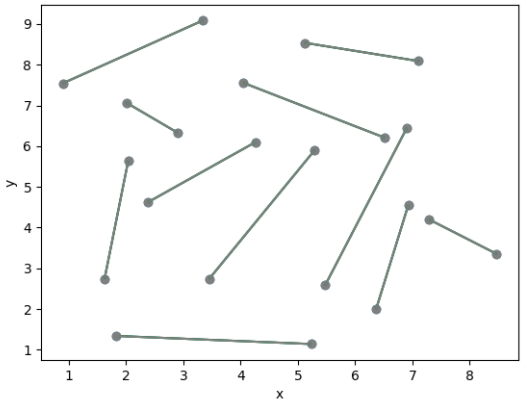
`is_intersectiom`: `True`



Rysunek 3: Ilustracja wyniku `find_intersections` dla zbioru A

3.3 Zbiór C

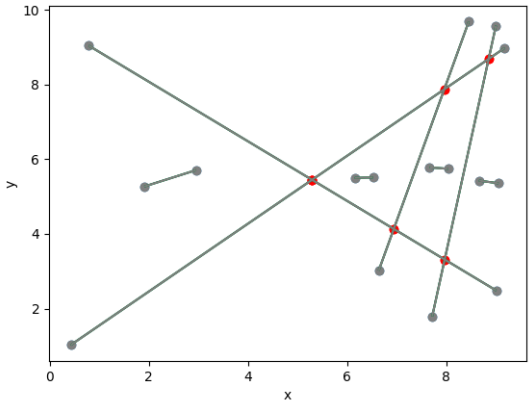
Zbiór testuje czy algorytm poprawnie radzi sobie w przypadku, w którym nie ma przecięć.
Wyniki:
`is_intersectiom`: `False`



Rysunek 4: Ilustracja wyniku `find_intersections` dla zbioru A

3.4 Zbiór D

Zbiór testuje czy algorytm poprawnie radzi sobie w przypadku, w którym przecięcia wykrywane są więcej niż jeden raz.
Wyniki:
`is_intersectiom`: `True`



Rysunek 5: Ilustracja wyniku `find_intersections` dla zbioru A

4 Podsumowanie

Zaimplementowane algorytmy przechodzą wybrane przeze mnie testy oraz te zawarte w narzędziu przygotowanym przez koło naukowe Bit. Wygenerowane przez program wyniki oraz funkcjonalność losowania lub zadawania własnych zbiorów jest zawarta w załączonym kodzie. Wraz ze sprawozdaniem załączone są także wygenerowane przez program animacje ilustrujące działanie algorytmu dla wybranych zbiorów testowych.