Algorytmy Geometryczne,
Ćwiczenie 4 – Przecinanie się odcinków

Maciej Kmąk

Grudzień 2024

1 Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z wykrywaniem przecięć odcinków w przestrzeni 2D oraz implementacją algorytmu zamiatania. Szczególną uwagę poświęcono strukturze stanu ("stan miotły"), pozwalającej sprawdzić, czy choć jedna para odcinków w zadanym zbiorze się przecina. Następnie skupiono się na zmianie struktury zdarzeń, aby umożliwić detekcję wszystkich przecięć odcinków, w tym podanie ich liczby, współrzędnych oraz identyfikację odcinków, które się przecinają. Ćwiczenie pozwoliło na przeanalizowanie i dostosowanie algorytmu do różnych zestawów danych i wymagań wyniku.

2 Dane techniczne

Program (w pliku „kmak_kod_4.ipynb") jest napisany w języku Python w środowisku Jupyter Notebook. Wykresy punktów i wizualizacje w formacie GIF powstały dzięki narzędziu przygotowanemu przez koło naukowe Bit, które korzysta z biblioteki `matplotlib`. Narzędzia dostępne w bibliotece `matplotlib` i `tkinter` umożliwiły zadawanie odcinków przy pomocy myszki. Do generowania punktów została użyta funkcja z biblioteki `NumPy - random.uniform()`. Przedstawiony program został wyegzekwowany na komputerze z procesorem Intel® Core™ i5-1235U z systemem operacyjnym Windows 11 Pro.

3 Wstęp teoretyczny

Definicja problemu:

1. Określenie, czy istnieje przynajmniej jedna para odcinków, które się przecinają.
2. Znalezienie wszystkich punktów przecięć i odcinków zaangażowanych w te przecięcia.

Rozwiązania problemu opierają się na algorytmach geometrycznych, które mogą być bardziej lub mniej efektywne w zależności od liczby odcinków i przecięć w zbiorze danych.

Algorytm zamiatania: Jednym z zaawansowanych podejść jest metoda zamiatania, która zamiast analizować każdą parę odcinków (metoda brutalna, złożoność $O(n^2)$) wykorzystuje dynamiczną strukturę danych do optymalizacji obliczeń. Algorytm działa w następujący sposób:

1. **Miotła:** Prosta przesuwana wzdłuż osi O_x, która analizuje odcinki w miarę ich przecinania.
2. **Struktura zdarzeń ($\boldsymbol{Q}$):** Zawiera uporządkowane zdarzenia, takie jak początki, końce i punkty przecięć odcinków.
3. **Struktura stanu ($\boldsymbol{T}$):** Przechowuje aktywne odcinki przecinające miotłę, uporządkowane według współrzędnych y.

Założenia algorytmu zamiatania:

- Żaden odcinek nie jest pionowy
- Dwa odcinki przecinają się w co najwyżej jednym punkcie
- Żadne trzy odcinki nie przecinają się w jednym punkcie

Zalety algorytmu zamiatania:

- Złożoność obliczeniowa:
  $O((P+n)\cdot log(n))$, gdzie P to liczba przecięć, a n to liczba odcinków.
- Efektywność dla zbiorów z niewielką liczbą przecięć w stosunku do liczby odcinków.

Algorytm ten pozwala nie tylko wykryć, czy odcinki się przecinają, ale również znaleźć wszystkie punkty przecięć oraz analizować ich rozmieszczenie. Jest szeroko stosowany w aplikacjach wymagających szybkich i dokładnych analiz geometrycznych.

Opis algorytmu zamiatania:

```
1. Utwórz pustą strukturę stanu T(np. wzbogacone, zrównoważone drzewo poszukiwań).
2. Utwórz strukturę zdarzeń Q, wstawiając do niej wszystkie:
   - Początki odcinków (lewe końce).
   - Końce odcinków (prawe końce).
   - Punkty przecięć wszystkich par, które kiedykolwiek były sąsiadami.
   Posortuj Q względem współrzędnych x.
3. Powtarzaj, dopóki Q nie jest puste:
   a. Pobierz pierwsze zdarzenie p z Q.
   b. Jeśli p jest początkiem odcinka s:
      i. Dodaj s do T.
      ii. Znajdź sąsiadów s w T: s' (powyżej) i s'' (poniżej).
      iii. Jeśli s' przecina s w punkcie p1:
          - Dodaj p1 do Q.
      iv. Jeśli s przecina s'' w punkcie p2:
          - Dodaj p2 do Q.
   c. Jeśli p jest końcem odcinka s:
      i. Usuń s z T.
      ii. Znajdź sąsiadów s w T: s' (powyżej) i s'' (poniżej).
      iii. Jeśli s' przecina s'' w punkcie p3:
          - Dodaj p3 do Q, jeśli jeszcze go tam nie ma.
   d. Jeśli p jest punktem przecięcia odcinków s i s':
      i. Zamień kolejność s i s' w T.
      ii. Znajdź sąsiadów s i s' w T po zamianie:
          - Jeśli s przecina sąsiada w w punkcie p4:
            - Dodaj p4 do Q, jeśli jeszcze go tam nie ma.
          - Jeśli s' przecina sąsiada v w punkcie p5:
            - Dodaj p5 do Q, jeśli jeszcze go tam nie ma.
4. Algorytm kończy działanie, gdy Q jest puste.
   Wynikiem są wszystkie punkty przecięć znalezione w trakcie pracy algorytmu.
```

Opis algorytmu zamiatania.

Początkowo zaimplementowano uproszczony algorytm zamiatania, którego celem było jedynie sprawdzenie, czy w zbiorze odcinków istnieje przynajmniej jedna para odcinków przecinających się. Ten algorytm charakteryzował się prostotą i był efektywny w wykrywaniu pierwszego przecięcia, dzięki czemu unikał zbędnych obliczeń. W kolejnych iteracjach został rozszerzony o możliwość wykrywania wszystkich przecięć, co wymagało modyfikacji struktury zdarzeń i bardziej zaawansowanego zarządzania stanem.

4 Zbiory i przebieg doświadczenia

4.1 Przebieg doświadczenia

Początkowo algorytm zamiatania został zaimplementowany w celu sprawdzenia, czy istnieje przynajmniej jedna para odcinków, które się przecinają, bez wykrywania wszystkich punktów przecięć. Rozwiązanie wykorzystuje dwie kluczowe struktury danych: strukturę zdarzeń oraz strukturę stanu. Struktura zdarzeń to posortowana lista obiektów typu `Event`, reprezentujących początki i końce odcinków względem współrzędnej x, co umożliwia przetwarzanie zdarzeń od lewej do prawej bez potrzeby dodawania nowych. Struktura stanu, zaimplementowana jako `SortedSet`, przechowuje aktywne odcinki przecinające aktualną pozycję "miotły", uporządkowane względem y. Podczas obsługi typów zdarzeń (`start` i `end`) algorytm sprawdza przecięcia między sąsiadami w strukturze stanu. Wykrycie przecięcia kończy działanie algorytmu, zwracając wynik `True`, natomiast w przypadku braku przecięć po przetworzeniu wszystkich zdarzeń zwracane jest `False`.

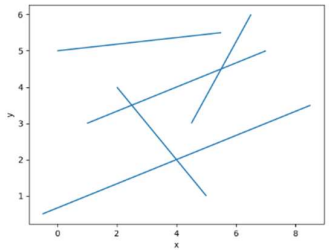
W następnym kroku implementacji algorytmu zamiatania zmieniono strukturę zdarzeń na kopiec priorytetowy (`heapq`), co umożliwiło bardziej efektywne zarządzanie zdarzeniami. Obiekty typu Event zostały wzbogacone o nowy typ zdarzenia: `intersection`, który reprezentuje punkt przecięcia dwóch odcinków. Kopiec priorytetowy sortuje zdarzenia według współrzędnej x, a w przypadku równych wartości x, priorytet ustalany jest na podstawie typu zdarzenia (`start > intersection > end`).

Dzięki tym zmianom możliwe było dynamiczne dodawanie zdarzeń przecięcia do struktury zdarzeń w trakcie działania algorytmu, co pozwala wykrywać wszystkie punkty przecięć. W momencie przetwarzania zdarzenia `intersection`, algorytm aktualizuje porządek odcinków w strukturze stanu oraz sprawdza nowe sąsiedztwa, co umożliwia poprawne obsłużenie kolejnych zdarzeń przecięcia. Te modyfikacje znacząco zwiększyły funkcjonalność algorytmu, pozwalając na znalezienie wszystkich przecięć w czasie $O((P + n) \cdot log(n))$, gdzie P to liczba przecięć, a n to liczba odcinków.

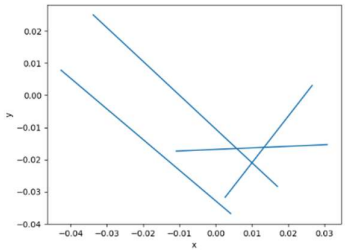
Uzupełniono procedury algorytmu, aby generowały pliki w formacie .GIF, wizualizujące proces wykrywania przecięć odcinków. Dla przedstawionych zbiorów danych (sekcja 4.2 Testowane zbiory danych) pliki te ilustrują kolejne etapy działania algorytmu, w tym obsługę zdarzeń, aktualizację struktury stanu oraz identyfikację punktów przecięć. Wizualizacje pozwalają lepiej zrozumieć dynamikę i działanie algorytmu. Ze względu na ograniczenia formatu .pdf, pliki .GIF nie zostały załączone w sprawozdaniu, ale są dostępne w folderze kmak_gifs.zip, przesłanym wraz ze sprawozdaniem. Folder zawiera również legendę wyjaśniającą oznaczenia użyte w wizualizacjach.

W celu weryfikacji działania programu wprowadzono układ odcinków, w którym pewne przecięcia mogłyby być wykrywane więcej niż jeden raz. Program uwzględnia takie sytuacje dzięki wykorzystaniu zbioru `checked_pairs`, który przechowuje informacje o już sprawdzonych parach odcinków. Każda para odcinków, zanim zostanie dodana do struktury zdarzeń jako punkt przecięcia, jest weryfikowana pod kątem istnienia w zbiorze `checked_pairs`. Jeśli dana para odcinków została już sprawdzona, przecięcie nie jest ponownie obliczane. Dzięki temu algorytm unika wielokrotnego wykrywania i przetwarzania tych samych punktów przecięć, co zapewnia poprawne i efektywne działanie programu.

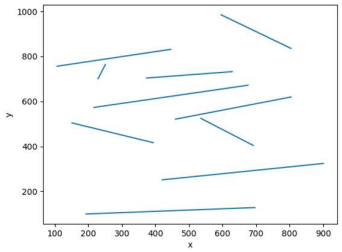
4.2 Testowane zbiory danych



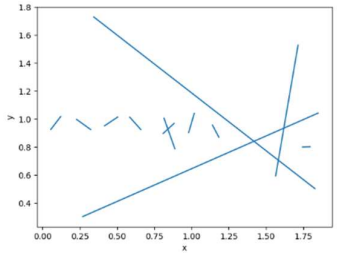
Wykres 4.2.1 Zbiór A



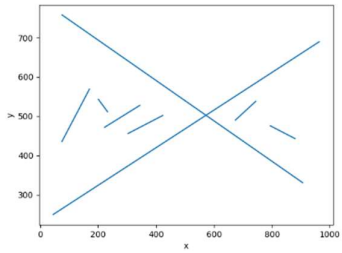
Wykres 4.2.2 Zbiór B



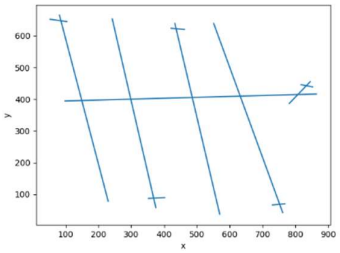
Wykres 4.2.3 Zbiór C



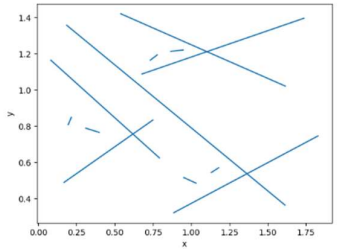
Wykres 4.2.4 Zbiór D



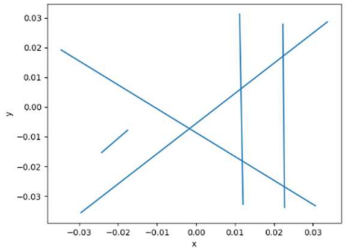
Wykres 4.2.5 Zbiór E



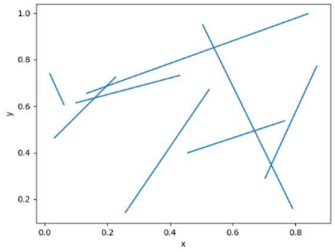
Wykres 4.2.6 Zbiór F



Wykres 4.2.7 Zbiór G



Wykres 4.2.8 Zbiór H



Wykres 4.2.9 Zbiór I

Zbiory przedstawione na wykresach(Wykresy 4.2.1-4.2.9) obrazują wybrane przypadki testowe dla algorytmu wykrywania przecięć odcinków:

- **Zbiory A i B** (Wykresy 4.2.1 i 4.2.2): To podstawowe sytuacje kontrolne, pozwalające zweryfikować poprawność działania algorytmu w prostych układach odcinków.
- **Zbiór C** (Wykres 4.2.3): Nie zawiera punktów przecięcia, co umożliwia przetestowanie działania algorytmu w przypadku, gdy przecięcia nie występują.
- **Zbiory D i E** (Wykresy 4.2.4 i 4.2.5): Przedstawiają sytuacje, w których kilkakrotnie mogłoby być wykrywane centralne przecięcie odcinków, co pozwala sprawdzić mechanizm eliminacji duplikatów.
- **Zbiory F, G i H** (Wykresy 4.2.6 - 4.2.8): To bardziej skomplikowane układy odcinków, zawierające trudne do analizy przypadki.
- **Zbiór I** (Wykres 4.2.9): Został wygenerowany przy użyciu funkcji generującej odcinki i początkowo sprawiał problemy implementacyjne, wynikające z niestandardowego rozkładu odcinków oraz ich nietypowego położenia.

Każdy z tych zbiorów danych umożliwia przetestowanie różnych aspektów działania algorytmu, od prostych sytuacji kontrolnych po bardziej złożone przypadki implementacyjne.

5 Wyniki doświadczenia

5.1 Wyniki algorytmu sprawdzającego istnienie przynajmniej jednej pary odcinków, przecinających się

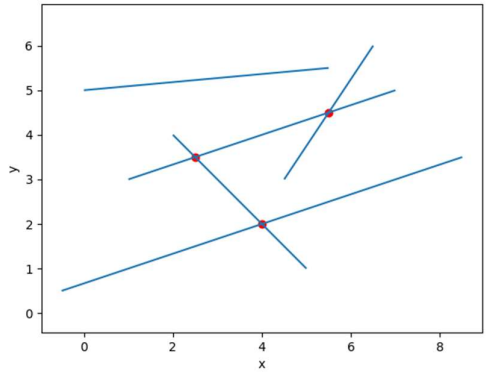
Wyniki działania przedstawiono w tabeli (Tabela 5.1) poniżej. Dla wszystkich zbiorów odcinków przedstawionych na wykresach 4.2.1-4.2.9, z wyjątkiem zbioru C, algorytm zwrócił wartość `True`, co oznacza, że wykryto przecięcie odcinków. Natomiast dla zbioru C algorytm zwrócił wartość `False`, co było zgodne z oczekiwaniami, ponieważ zbiór ten nie zawierał żadnych punktów przecięcia. Dzięki temu algorytm potwierdził swoją poprawność w obsłudze zarówno przypadków z przecięciami, jak i bez nich.

Tabela 1. Wyniki działani algorytmu sprawdzającego czy występuje przynajmniej jedno przecięcie odcinków

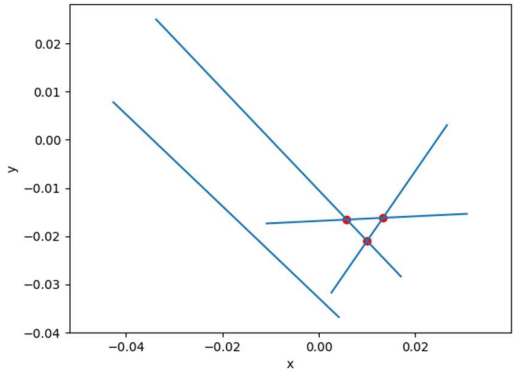
Zbiór A	Zbiór B	Zbiór C	Zbiór D	Zbiór E	Zbiór F	Zbiór G	Zbiór H	Zbiór I
`True`	`True`	`False`	`True`	`True`	`True`	`True`	`True`	`True`

5.2 Wyniki algorytmu wyznaczającego wszystkie pary odcinków, przecinających się

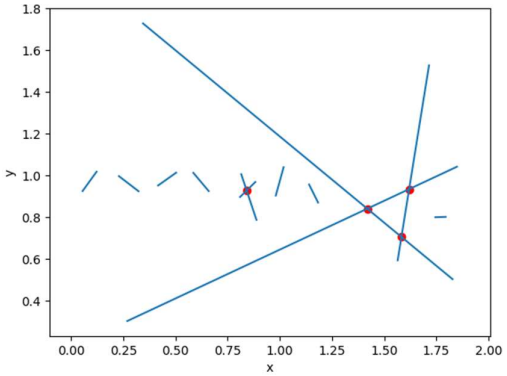
Poniżej na wykresach (Wykresy 5.3.1-5.3.8, poza 5.3.3) przedstawiono wyniki działania algorytmu wyznaczającego punkty przecięcia odcinków dla zbiorów A, B, D, E, F, G, H oraz I (widocznych na wykresach 4.2.1-4.2.9 za wyjątkiem 4.2.3). Zbiór C (Wykres 4.2.3) został pominięty, ponieważ nie zawierał punktów przecięcia. Dla spójności oznaczeń Wykres 5.3.3 został pominięty.



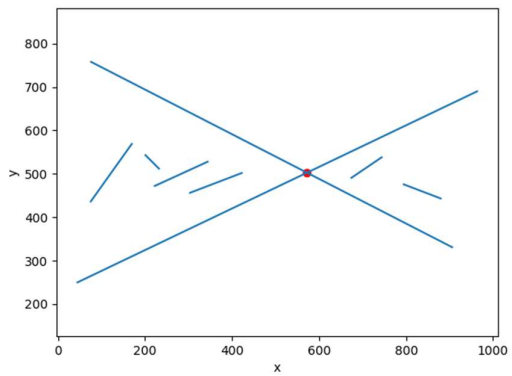
Wykres 5.2.1 Zbiór A



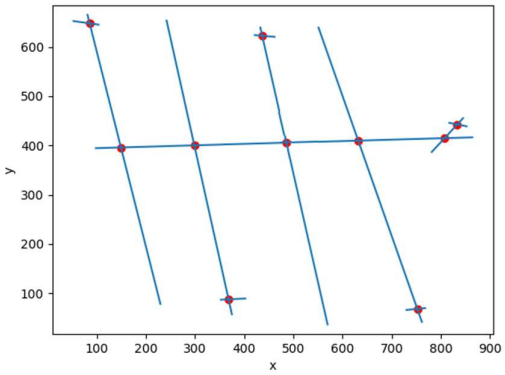
Wykres 5.2.2 Zbiór B



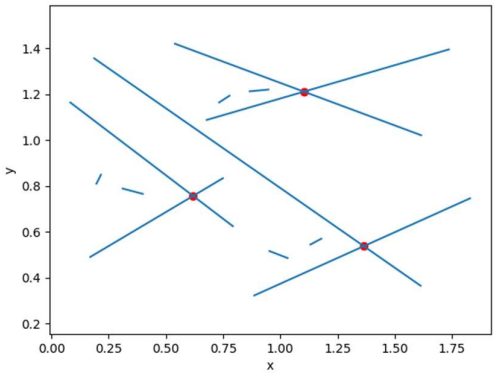
Wykres 5.2.3 Zbiór D



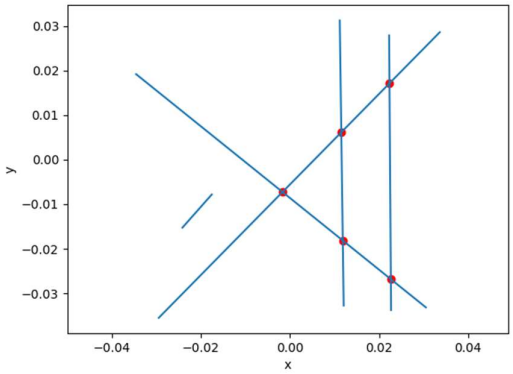
Wykres 5.2.2 Zbiór E



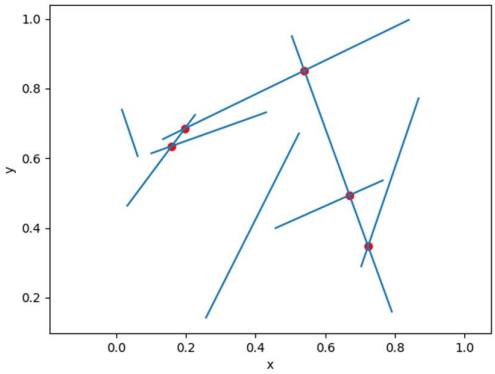
Wykres 5.2.1 Zbiór F



Wykres 5.2.2 Zbiór G



Wykres 5.2.1 Zbiór H



Wykres 5.2.2 Zbiór I

Na podstawie wyników działania algorytmu wyznaczającego punkty przecięcia odcinków (Wykresy 5.2.1-5.2.8) można wyciągnąć następujące wnioski:

1. **Poprawność dla prostych zbiorów (A i B):**
   Wyniki na wykresach 5.2.1 i 5.2.2 potwierdzają poprawność algorytmu w przypadku prostych układów odcinków.

2. **Brak przecięć w zbiorze C:**
   Zbiór C (Wykres 4.2.3) został pominięty w tej części analizy (nie pokazano wykresu 5.2.3), ponieważ algorytm poprawnie zwrócił wynik wskazujący brak punktów przecięcia = pusta lista.

3. **Poprawność dla przypadków z centralnym przecięciem (D i E):**
   Na wykresach 5.2.3 i 5.2.4 przedstawiono zbiory zawierające odcinki z centralnym przecięciem. Algorytm skutecznie wykrył wszystkie punkty przecięcia.

4. **Analiza złożonych przypadków (F, G, H i I):**

   - **Zbiór F** (Wykres 5.2.5): Algorytm poprawnie zidentyfikował przecięcia w układzie z równoległymi i blisko położonymi odcinkami.

   - **Zbiór G** (Wykres 5.2.6): Skutecznie wykryto przecięcia w bardziej rozbudowanym układzie odcinków, co potwierdza jego działanie w asymetrycznych przypadkach.

- **Zbiór H** (Wykres 5.2.7): Szczególnie trudny przypadek zawierający niemal pionowe odcinki został poprawnie obsłużony. Algorytm skutecznie uwzględnił punkty przecięcia bez pomijania żadnych istotnych zdarzeń.
- **Zbiór I** (Wykres 5.2.8): Układ odcinków wygenerowany funkcją `uniform`. Algorytm wykrył wszystkie punkty przecięcia i prawidłowo je sklasyfikował.

Podsumowanie:
Algorytm wyznaczania przecięć odcinków w zbiorze działa poprawnie zarówno w prostych, jak i złożonych przypadkach, niezależnie od rozkładu odcinków i ich orientacji. W podstawowych testowych układach, takich jak zbiory A i B, algorytm poprawnie identyfikuje punkty przecięcia, co potwierdza jego skuteczność w prostych geometrach. W bardziej złożonych przypadkach, takich jak zbiory F, G, H i I, algorytm zachowuje swoją efektywność, poprawnie wykrywając punkty przecięcia nawet w trudnych układach, gdzie odcinki są blisko siebie, asymetryczne lub niemal pionowe. Działanie algorytmu zostało również zweryfikowane na losowo generowanym zbiorze I. Kluczową rolę w poprawności wyników odgrywa mechanizm unikania wielokrotnego sprawdzania i dodawania tych samych par odcinków (`checked_pairs`), który skutecznie eliminuje duplikaty przecięć, zapewniając niezawodność nawet w najbardziej złożonych konfiguracjach.

Ze względu na ograniczenia formatu .pdf, pliki .GIF prezentujące działanie obu algorytmów, zarówno algorytmu wyznaczania jednego przecięcia, jak i algorytmu wykrywającego wszystkie punkty przecięcia, zostały dołączone w folderze kmak_gifs.zip, przesłanym wraz ze sprawozdaniem. Pliki te wizualizują kolejne kroki działania algorytmów, w tym przetwarzanie zdarzeń, aktualizację struktury stanu, sprawdzanie przecięć sąsiednich odcinków, a także aktualne i poprzednie położenie "miotły" podczas przetwarzania zdarzeń.

Legenda dla wizualizacji GIF przecięć odcinków:

- Punkty niebieskie: Punkty początkowe odcinków, oznaczone dla identyfikacji podczas analizy.
- Punkty jasnoniebieskie: Punkty końcowe odcinków, wskazujące zakończenie segmentu.
- **Czarne** odcinki: Początkowe segmenty, oznaczone na samym początku wizualizacji.
- Pomarańczowa linia pionowa: Pozycja miotły w trakcie przetwarzania bieżącego zdarzenia.
- Zielone odcinki: Odcinki które kiedykolwiek znalazły się w strukturze stanu
- Żółte odcinki: Segmenty, które są aktualnie rozpatrywane
- Czerwone odcinki: Odcinki, które mają przecięcia w bieżącym kroku obliczeń.
- Czerwone punkty: Punkty przecięcia odcinków, dodane do wyniku końcowego.
- Szare pionowe linie: Historia pozycji miotły w poprzednich krokach, użyta do śledzenia dynamiki procesu.

6 Wnioski

Na podstawie przeprowadzonych testów i analizy działania algorytmów wykrywających przecięcia odcinków można sformułować następujące wnioski:

1. **Poprawność algorytmu sprawdzającego istnienie przynajmniej jednego przecięcia odcinków:**
   Algorytm skutecznie identyfikował przypadki, w których występowało co najmniej jedno przecięcie odcinków. Dla prostych układów (np. zbiory A i B) oraz bardziej złożonych konfiguracji (np. zbiory F, G, H i I) algorytm zwracał poprawne wyniki. Zbiór C, w którym brakowało przecięć, został prawidłowo oznaczony jako przypadek bez przecięć, co potwierdza jego poprawność i niezawodność.
2. **Poprawność algorytmu wykrywającego wszystkie punkty przecięcia odcinków:** Algorytm poprawnie wyznaczał wszystkie punkty przecięcia dla analizowanych zbiorów, niezależnie od liczby odcinków, ich rozkładu czy orientacji. Mechanizm eliminacji duplikatów (zbiór `checked_pairs`) skutecznie zapobiegał wielokrotnemu dodawaniu tych samych przecięć do wyników, co szczególnie dobrze sprawdziło się w przypadkach z „centralnymi" przecięciami (np. zbiory D i E).

Algorytmy działały niezawodnie w bardziej skomplikowanych układach, takich jak zbiór H z niemal pionowymi odcinkami oraz zbiór I, który został wygenerowany losowo funkcją `uniform`.

7 Podsumowanie zagadnienia

Przeprowadzone testy potwierdziły poprawność i efektywność zaimplementowanych algorytmów wykrywających przecięcia odcinków, zarówno w przypadku sprawdzania jednego przecięcia, jak i wyznaczania wszystkich punktów przecięcia.

- **Algorytm sprawdzania istnienia przynajmniej jednego przecięcia odcinków** okazał się niezawodny, poprawnie identyfikując układy z przecięciami i bez przecięć, co zapewniło solidną podstawę dla bardziej zaawansowanych operacji.
- **Algorytm wykrywający wszystkie punkty przecięcia** skutecznie znajdował i klasyfikował punkty przecięcia w różnych układach odcinków, eliminując duplikaty i zapewniając precyzyjne wyniki nawet w trudnych przypadkach geometrycznych.
- **Efektywność działania algorytmów** była widoczna zarówno w prostych, jak i złożonych układach, niezależnie od liczby odcinków, ich rozkładu czy orientacji.

Podsumowując, zaimplementowane algorytmy umożliwiły skuteczną identyfikację przecięć odcinków, a ich wyniki były zgodne z założeniami teoretycznymi i przewidywaniami. Dzięki temu mogą być one wykorzystane do rozwiązywania problemów geometrycznych w szerokim zakresie zastosowań.

8 Zadanie dodatkowe

Celem implementacji alternatywnej struktury stanu było zbadanie wpływu różnych metod zarządzania aktywnymi odcinkami na efektywność algorytmu wykrywania przecięć. Pierwotnie wykorzystana struktura `SortedSet` automatycznie utrzymywała porządek odcinków według współrzędnej y, co upraszczało obsługę zdarzeń, ale mogło wprowadzać dodatkowy narzut na czas działania. W alternatywnym rozwiązaniu struktura stanu została zaimplementowana jako lista, w której porządek elementów był zarządzany ręcznie za pomocą wyszukiwania binarnego i dynamicznej aktualizacji.

Wyniki tego doświadczenia przedstawiono w tabeli(Tabela 8.1):

Tabela 8.1. Wyniki testów wydajnościowych

Liczba odcinków	**`find_intersections` [s]**	**`find_intersections_list` [s]**
50	0.015	0.008
100	0.034	0.024
500	2.798	0.608
1000	21.800	2.912

Wyniki testów pokazują, że lista zarządzana ręcznie za pomocą wyszukiwania binarnego przewyższa wydajnością strukturę `SortedSet` w przypadku dużych zestawów danych, co wynika z mniejszego narzutu operacyjnego. Różnice w czasie działania wynikają głównie z charakterystyki zastosowanych struktur danych. `SortedSet` jest bardziej zaawansowaną strukturą, co wiąże się z większymi kosztami operacyjnymi, podczas gdy lista z wyszukiwaniem binarnym jest prostszym rozwiązaniem, które działa lepiej w tym specyficznym przypadku.