Laboratorium 4
Wyznaczanie przecięć odcinków

Łukasz Kwinta

1 Dane Techniczne

Procesor: AMD Ryzen 7 5700U
System operacyjny: Ubuntu 20.04 w środowisku WSL 2 na Windows 11 x64
Pamięć ram: 32 GB DDR4

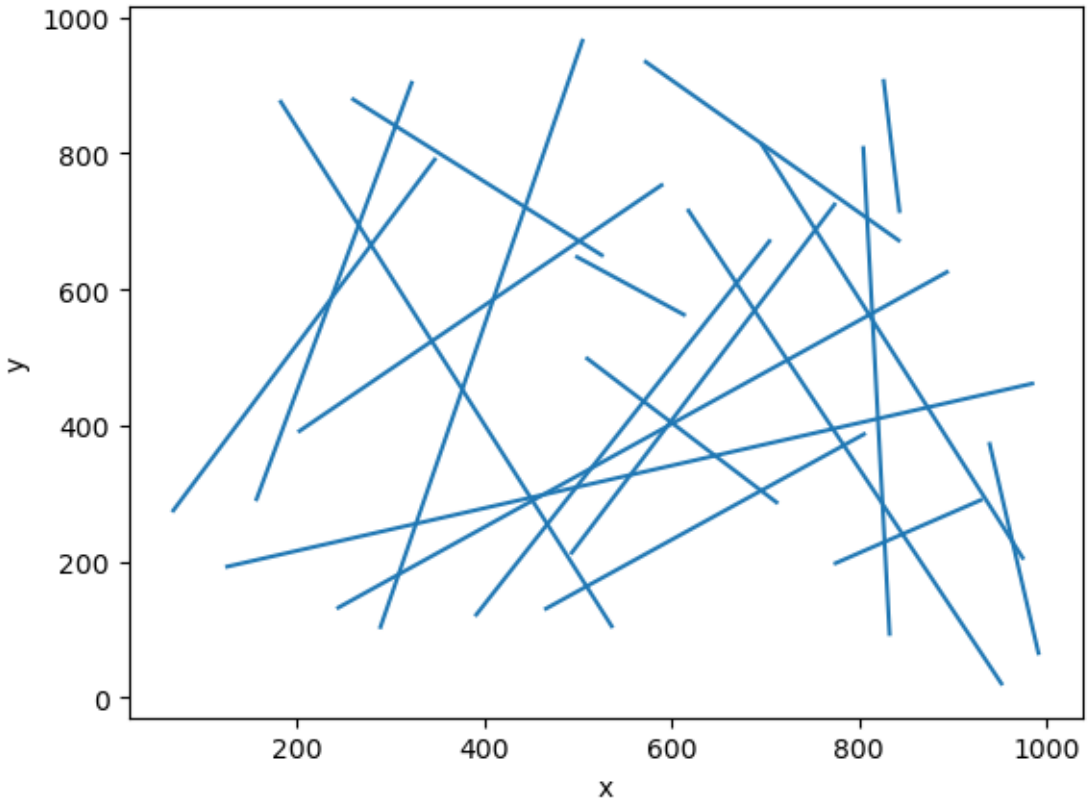
Środowisko i język: Python 3.9 + Jupyter Notebook w środowisku Anaconda
Wykresy tworzyłem przy pomocy narzędzia przygotowanego przez KN Bit, do obliczeń numerycznych używałem biblioteki numpy. Dane przechowywałem w zmiennych typu float – typ danych o rozmiarze 64 bitów, odpowiednik typu double w języku C.

2 Opis Realizacji Ćwiczenia

Celem ćwiczenia była implementacja algorytmu wyznaczającego przecięcia wszystkich odcinków na płaszczyźnie za pomocą algorytmu zamiatania.

2.1 Generowanie zbiorów testowych

Pierwszym krokiem realizacji ćwiczenia było przygotowanie funkcji generujacej losowe odcinki na płaszczyźnie. Wykorzystałem w tym celu funkcję wbudowaną w pakiety Pythona `random.random.uniform()` losującą floaty w zadanym przedziale. Gdy jakiś punkt wylosowany został pionowo to przesuwałem jeden jego koniec - aby zbiór danych nie zawierał pionowych odcinków.
Wygenerowany zbiór miał zawierać 20 odcinków w końcach o współrzędnych w przedziałach $x \in \langle 0, 1000 \rangle$ oraz $y \in \langle 0, 1000 \rangle$



Rysunek 1: Wizualizacja przykładowego zbioru wygenerowanych odcinków

2.2 Sprawdzenie czy w zbiorze prostych istnieje przecięcie

Kolejnym krokiem było zaimplementowanie uproszczonej wersji algorytmu który, kończy działanie w momencie znalezienia pierwszych prostych które się przecinają. Do reprezentacji struktury stanu i struktury zdarzeń użyłem `SortedSet` z bibliotek

`sortedcontainters` który jest realizacją drzewa BST pozwala dodawać i wyjmować elementy z drzewa w czasie logarytmicznym, oraz sprawdzanie ich poprzedników i następników.
Aby w łatwy sposób aktualizować kolejność odcinków w strukturze zdarzeń zaimplementowałem klasę reprezentującą odcinek w której zdefiniowałem własny operator porównania które oblicza wartość y danego odcinka dla danego x, oraz wykorzystałem statyczne pole do ustalania x aby wszystkie odcinki były porównywane w tym samym x. Wartość x aktualizuję podczas przetwarzania każdego zdarzenia.

Poniżej krótka lista kroków pokazująca działanie algorytmu:

1. Wstawienie krańców wierzchołków do struktury zdarzeń Q w porządku względem rosnących współrzędnych x
2. Dopóki są zdarzenia w Q:
   (a) Pobranie pierwsze zdarzenie z Q
   (b) Jeśli zdarzenie jest początkiem odcinka:
      i. Aktualizacja obecnego x do porównania odcinków
      ii. Dodanie odcinka do struktury stanu T
      iii. Jeśli odcinek przecina się z poprzednikiem lub następnikiem: Znaleziono przecięcie
   (c) Jeśli zdarzenie jest końcem odcinka:
      i. Aktualizacja obecnego x do porównania odcinków
      ii. Usunięcie odcinka który kończy ze struktury T
3. Koniec algorytmu - brak przecięcia

Do sprawdzanie przecięć użyłem prostego algorytmu korzystającego z wyznaczników. Na początek obliczyłem znaki 4 wyznaczników między odcinkami, a końcami drugiego odcinka korzystając z własnej funkcji znaku:

$$sgn = \begin{cases} 1 & \text{jeśli } x > \varepsilon \\ -1 & \text{jeśli } x < -\varepsilon \\ 0 & \text{w pozostałych przypadkach} \end{cases}$$

dla $\varepsilon = 10^{-12}$. A następnie jeśli znak któregoś wyznacznik był 0 to znaczy, że punkty które reprezentował są współliniowe, a więc koniec jednego z odcinków zawiera się w drugim. Na koniec sprawdzam, czy wyznaczniki są przeciwnych znaków, tzn. punkty sprawdzane leżą po przeciwnych stronach odcinka, z czego wynika, że istnieje przecięcie.

Jeśli już wiem, że przecięcie istnieje mogę wyznaczyć przecięcie korzystając z układu równań kierunkowych odcinku rozwiązanego metodą macierzy. Jeśli:

$$s_1 : y = a_1 x + b_1$$

$$s_2 : y = a_2 x + b_2$$

to punkt przecięcia $P = (x_P, y_P)$ ma wtedy współrzędne w następującym miejscu:

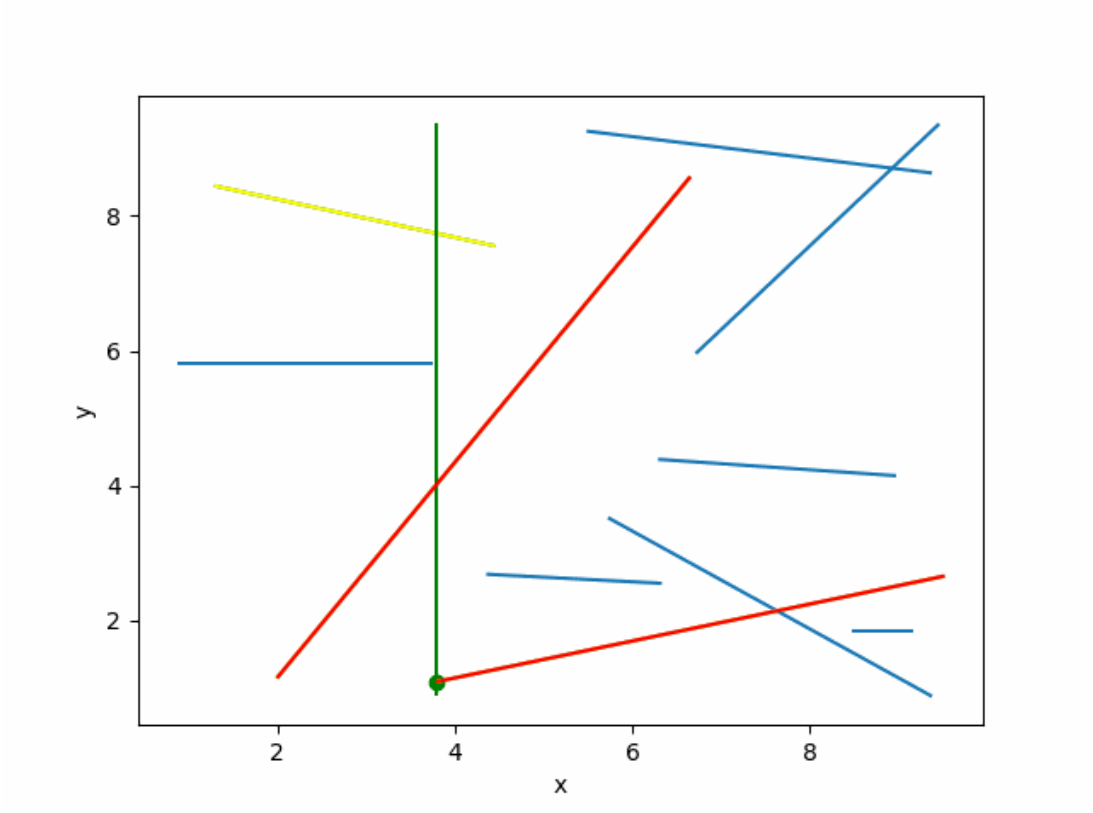
$$x_P = \frac{b_1 - b_2}{a_2 - a_1} \qquad y_P = \frac{a_2 b_1 - a_1 b_2}{a_2 - a_1}$$

2.3 Wizualizacja działania algorytmu sprawdzającego przecięcie

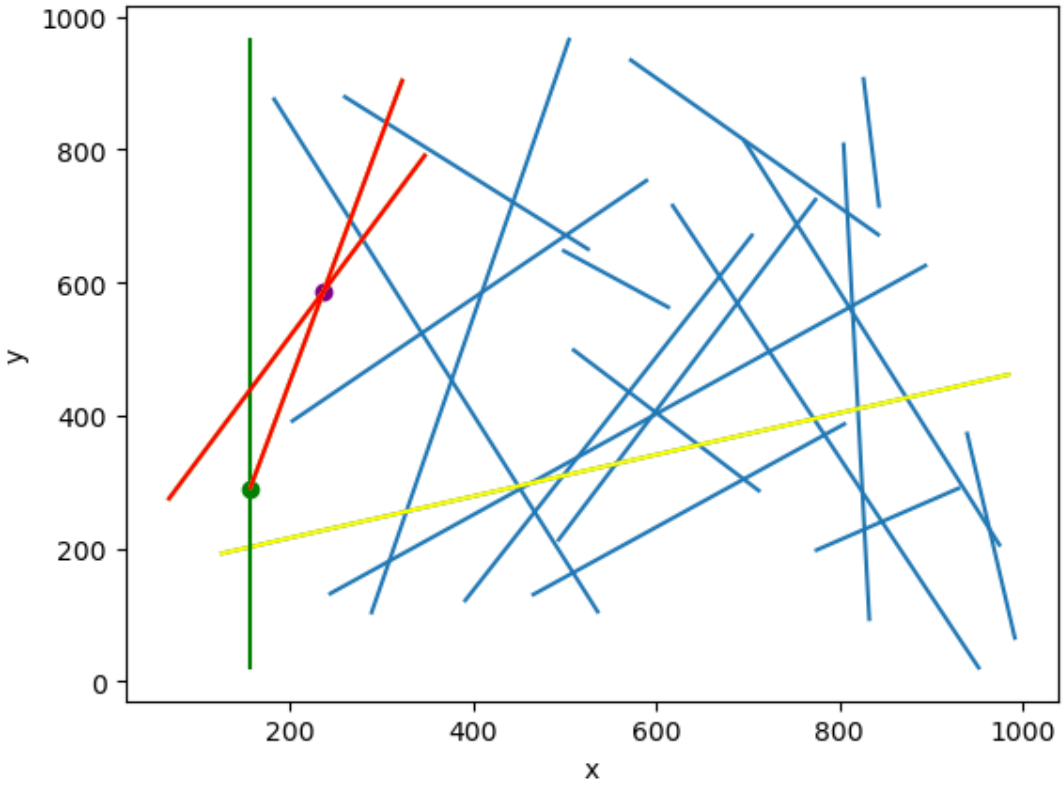
Zmodyfikowałem powyższy program tak aby generował animację z przebiegiem algorytmu. Przyjąłem następujące oznaczenia:

- Zielona prosta i zielony punkt - prosta wizualizuję miotłę, a punkt oznacza konkretny punkt zdarzenia który jest rozważany
- Żółte proste - proste znajdujące się w strukturze stanu T
- Czerwone proste - proste obecnie sprawdzane pod kątem przecinania się
- Fioletowy punkt - znaleziony punkt przecięcia

Poniżej zawarłem plik z animacją pokazujące kolejne kroki działania algorytmu oraz jego stan końcowy. ***Uwaga! Animacja może nie działać w każdym programie do plików PDF, została przetestowana w programie Adobe Acrobat gdzie działa. W razie gdyby animacja nie działała będzie wyświetlona jedna klatka animacji. Samą animację można znaleźć w notebooku z rozwiązaniem zadania.***



Rysunek 2: Animacja działania algorytmu dla przykładowego zbioru odcinków



Rysunek 3: Stan końcowy działania algorytmu dla losowo wygenerowanych odcinków

2.4 Algorytm wykrywający wszystkie przecięcia

Końcowym elementem wykonania zadania było stworzenie algorytmu który wykrywa wszystkie przecięcia w zbiorze prostych. Do realizacji struktury zdarzeń Q i struktury stanu T, użyłem `SortedSet` tak samo jak wcześniej lecz dla wygody zdarzenia musiałem opakować w obiekty klasy `Event` którą przygotowałem, dzięki nim wiem co oznacza dane zdarzenie i mogę przekazywać argumenty ze zdarzenia do zdarzenia.
Główną zmianą w strukturach danych była dodatkowa struktura zbioru `set()` realizowanego jako hash set, do sprawdzania które przecięcia zostały już dodane do struktury zdarzeń aby nie dodawać wielokrotnie tych samych przecięć do struktury zdarzeń.
Do zamiany punktów w strukturze stanu w momencie wykrywania przecięć wykorzystałem przygotowany mechanizm do ustalania x. Najpierw usuwam odcinki ze struktury stanu, następnie jako obecną wartość x ustawiam współrzędną x przecięcia które analizuję przesuniętą o $\varepsilon = 10^{-12}$.
Poniżej przedstawiam uproszczony schemat działania algorytmu:

1. Wstawienie krańców wierzchołków do struktury zdarzeń Q w porządku względem rosnących współrzędnych x
2. Dopóki są zdarzenia w Q:
   (a) Pobranie pierwsze zdarzenie z Q
   (b) Jeśli zdarzenie jest początkiem odcinka:
      i. Aktualizacja obecnego x do porównania odcinków
      ii. Dodanie odcinka do struktury stanu T
      iii. Jeśli odcinek przecina się z poprzednikiem lub następnikiem
         A. Sprawdzenie czy przecięcie jest w zbiorze przecięć
         B. Jeśli nie to dodanie przecięcia do Q
         C. Dodanie przecięcia do zbioru przecięć
   (c) Jeśli zdarzenie jest końcem odcinka:
      i. Aktualizacja obecnego x do porównania odcinków
      ii. Jeśli poprzednik i następnik odcinka się przecinają:
         A. Sprawdzenie czy przecięcie jest w zbiorze przecięć
         B. Jeśli nie to dodanie przecięcia do Q
         C. Dodanie przecięcia do zbioru przecięć
      iii. Usunięcie odcinka który kończy ze struktury T
   (d) Jeśli zdarzenie jest przecięciem

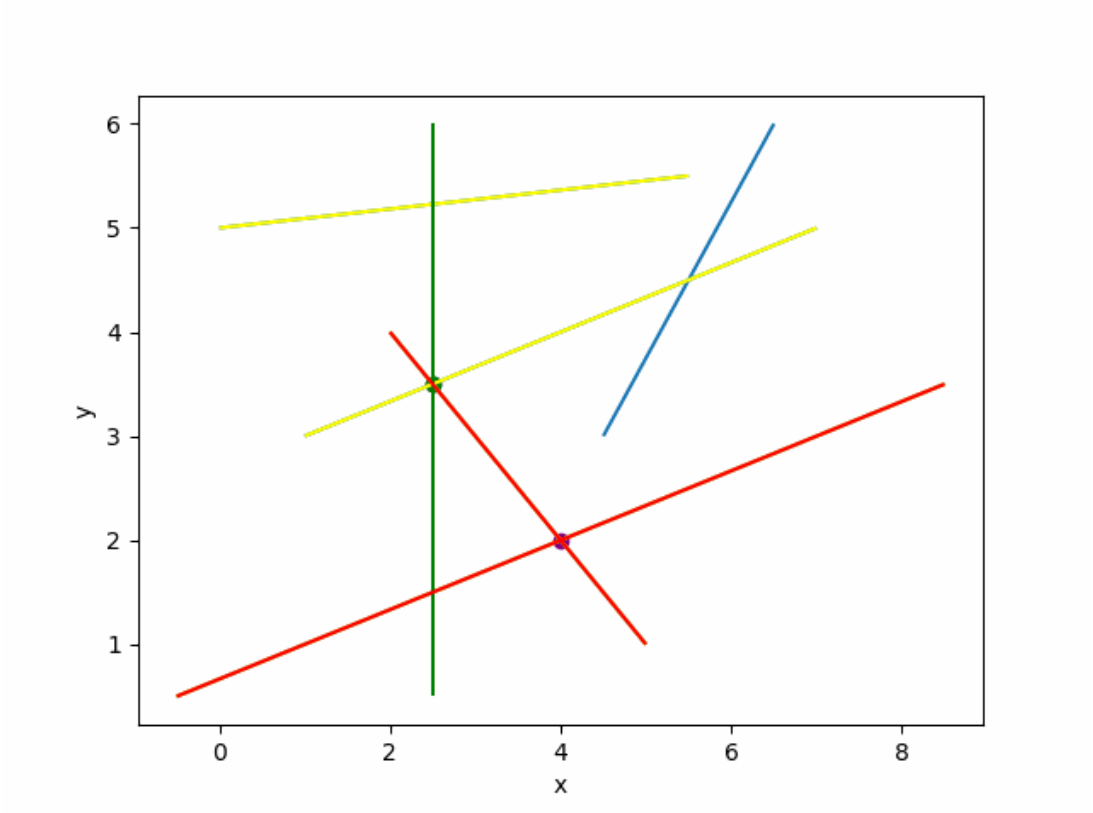
i. Niech s_1 i s_2 to odcinki które przecinają się w sprawdzanym punkcie (x_P, y_P)
ii. Usunięcie s_1 i s_2 ze struktury stanu T
iii. Aktualizacja obecnego x do porównania odcinków jako $x := x_P + \varepsilon$
iv. Dodanie s_1 i s_2 do struktury stanu T
v. Jeśli nowy sąsiad s_1 i s_1 się przecinają:
   A. Sprawdzenie czy przecięcie jest w zbiorze przecięć
   B. Jeśli nie to dodanie przecięcia do Q
   C. Dodanie przecięcia do zbioru przecięć
vi. Jeśli nowy sąsiad s_2 i s_2 się przecinają:
   A. Sprawdzenie czy przecięcie jest w zbiorze przecięć
   B. Jeśli nie to dodanie przecięcia do Q
   C. Dodanie przecięcia do zbioru przecięć

3. Zwrócenie listy przecięć

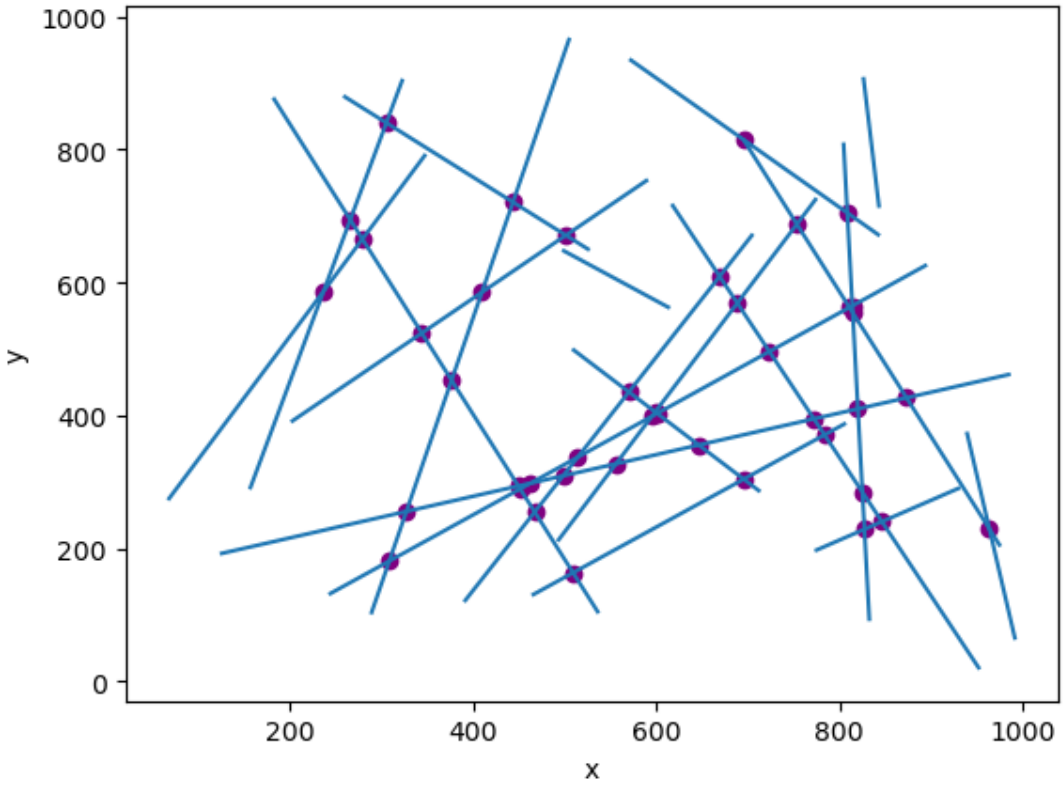
Przecięcia są sprawdzane i obliczane za pomocą tych samych funkcji co w poprzednim algorytmie.

2.5 Wizualizacja algorytmu wykrywającego wszystkie przecięcia

Wizualizację algorytmu przygotowałem podobnie jak w przypadku algorytmu sprawdzającego jedno przecięcie, wszystkie oznaczenia są identyczne. Podobnie jak wcześniej poniżej zawieram przykładową animację oraz efekt końcowy działania algorytmu dla losowo wygenerowanego zbioru odcinków.



Rysunek 4: Animacja działania algorytmu dla przykładowego zbioru odcinków



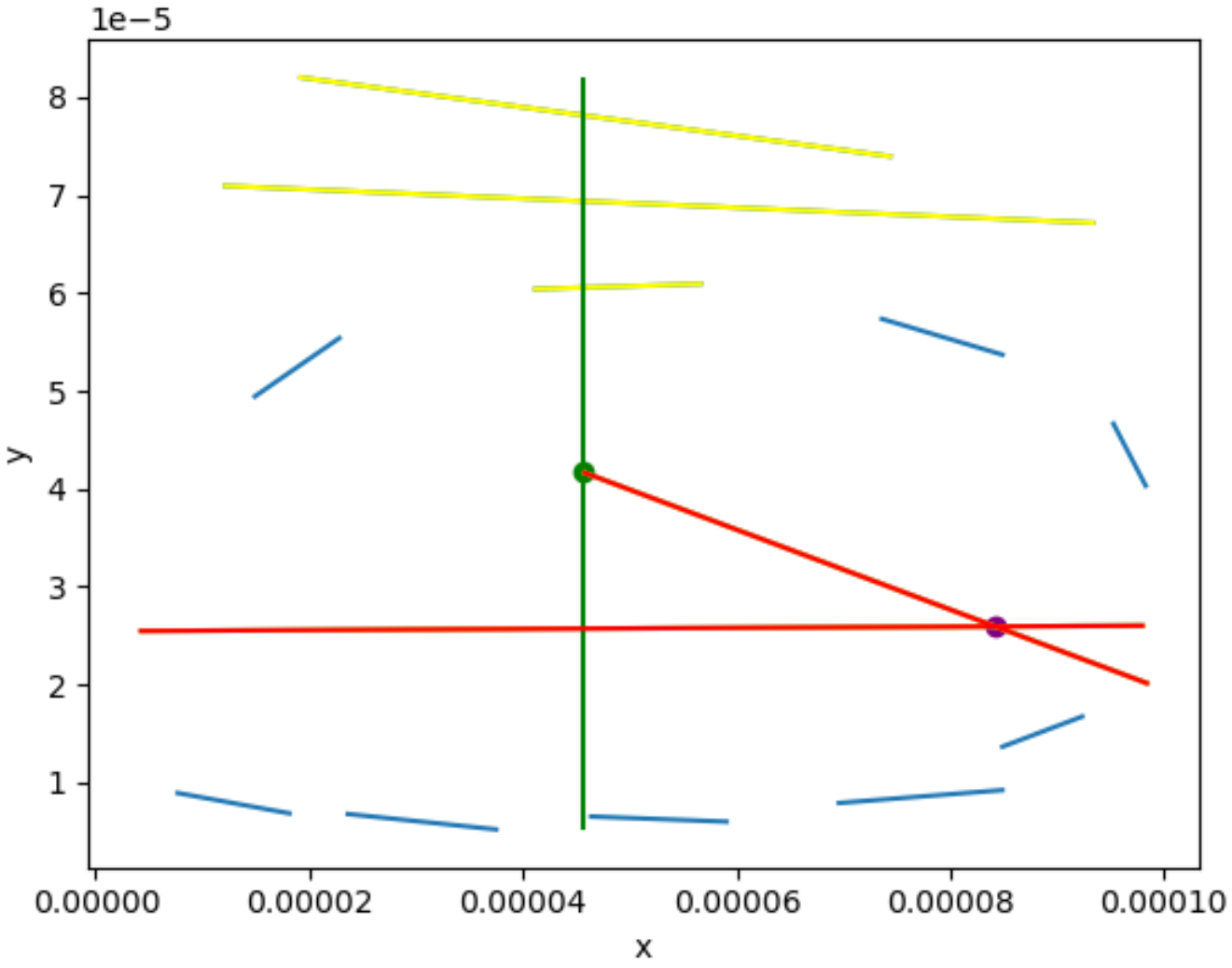
Rysunek 5: Stan końcowy działania algorytmu dla losowo wygenerowanych odcinków

3 Testy

3.1 Testy algorytmu szukającego jednego przecięcia

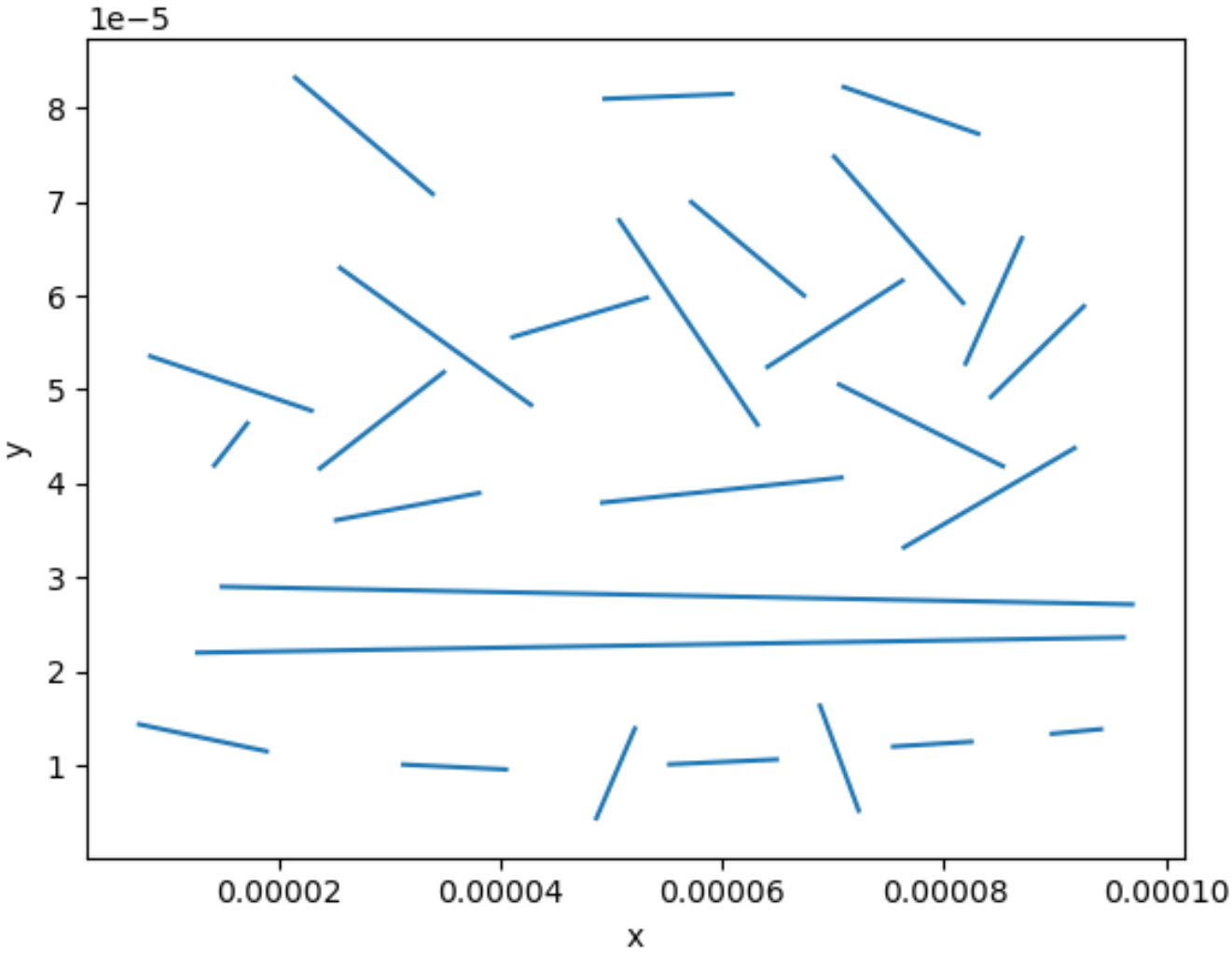
Wyróżniłem tutaj dwa przypadki szczególne które warto zawrzec w sprawozdaniu, więcej testów można obejrzeć w załączonym notebooku.

Pierwszym szczególnym przypadkiem to znalezienie przecięcia znajdującego się na końcu zbioru:



Rysunek 6: Wizualizacja wyniku szukania przecięć dla pierwszego szczególnego przypadku

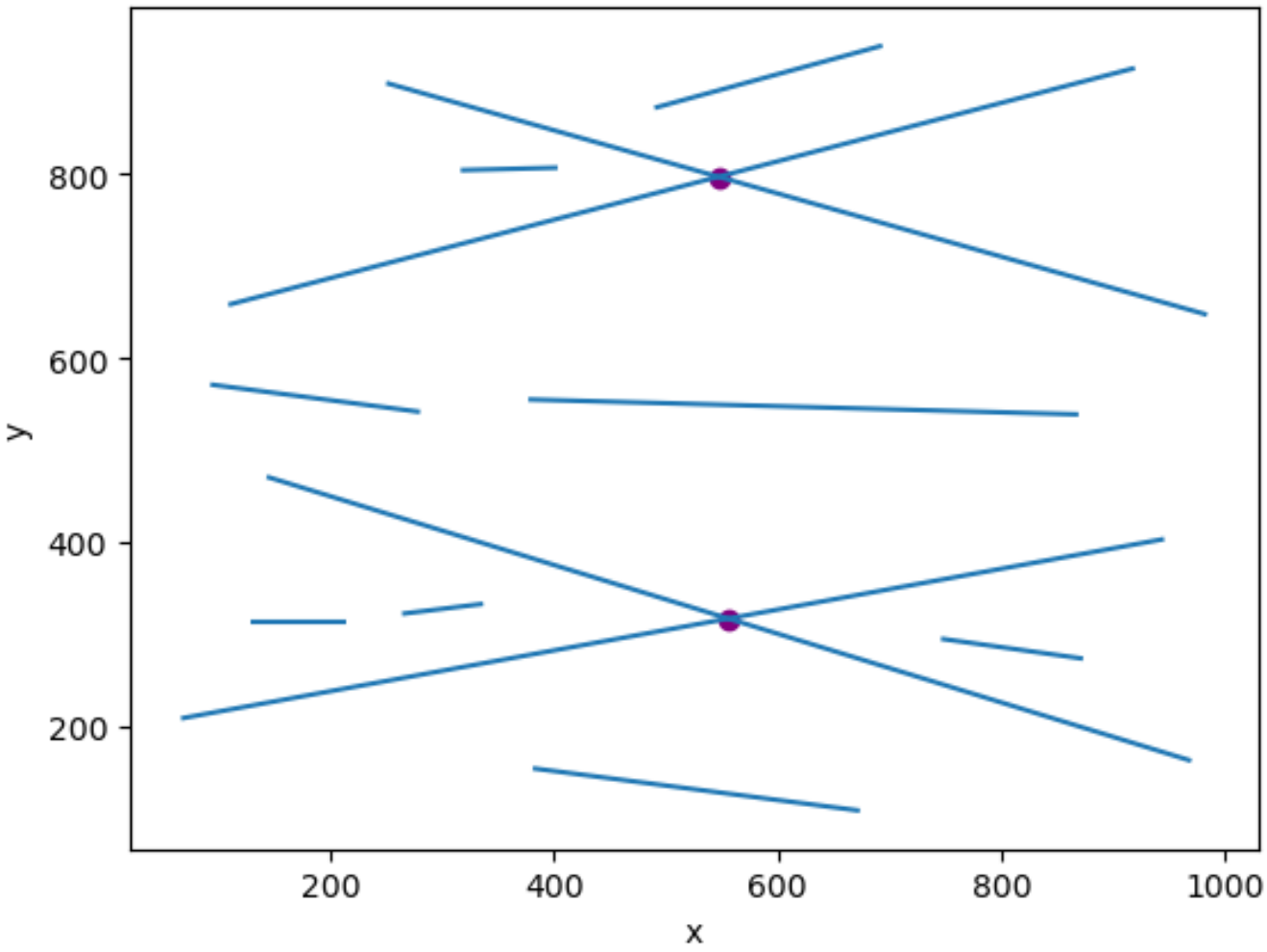
Drugim szczególnym przypadkiem to upewnienie się, że w gęstym zbiorze odcinków algorytm nie znajdzie, żadnego przecięcia:



Rysunek 7: Wizualizacja wyniku szukania przecięć dla pierwszego szczególnego przypadku

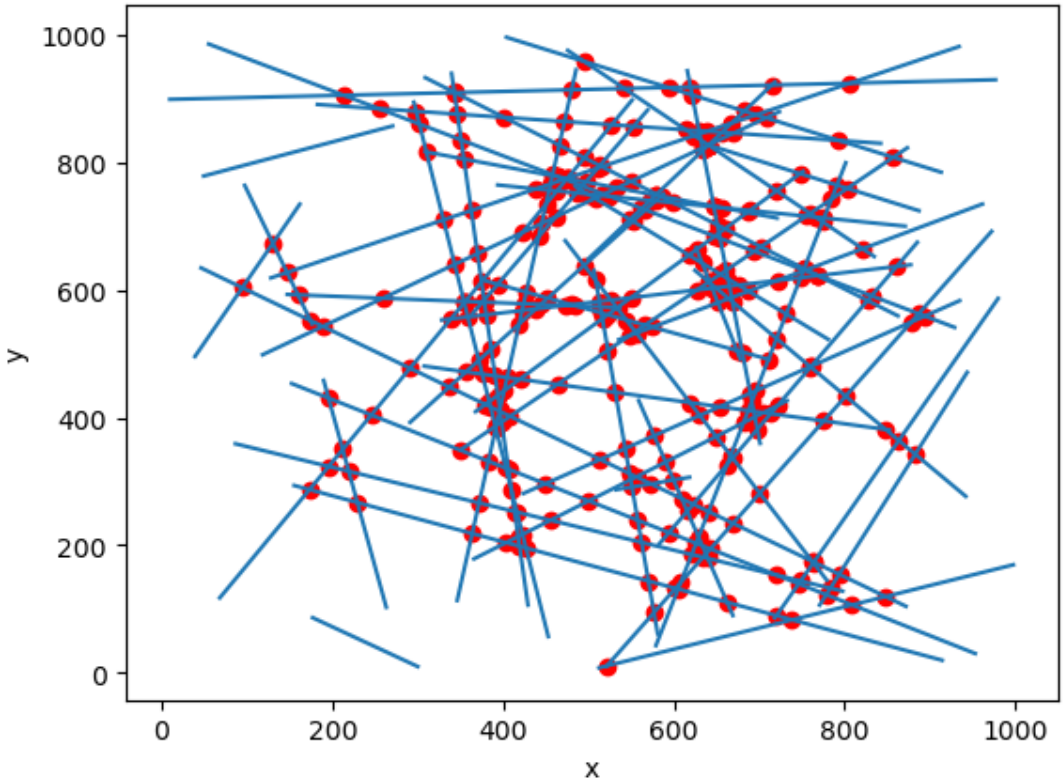
3.2 Testy algorytmu szukającego wszystkich przecięć

Przy testowaniu tego algorytmu główną trudnością były gęste zbiory odcinków dlatego zawarłem jeden taki przypadek w sprawozdaniu:



Rysunek 8: Wizualizacja wyniku szukania przecięć dla pierwszego szczególnego przypadku

Kolejnym ciekawym przypadkiem jest sprawdzenie czy algorytm poprawnie radzi sobie ze sprawdzaniem duplikatów, poniżej zamieszczam wynik działania algorytmu dla takiego testu. Test ten powoduje wielokrotne dodawanie tego samego przecięcia ponieważ jest kilka odcinków nie przecinających się między przecinającymi się odcinkami, a więc przy usuwaniu tychże odcinków sprawdzamy wiele razy te same przecinające się proste:



Rysunek 9: Wizualizacja wyniku szukania przecięć dla drugiego szczególnego przypadku

4 Wnioski

Na podstawie swoich testów mogę stwierdzić, że algorytmy działają poprawnie. Nie udało mi się wygenerować przypadków które nie działałyby poprawnie. Użycie `SortedSet` do struktur zdarzeń pozwoliło zaimplementować algorytm o odpowiedniej złożoności oraz pozwoliło zaoszczędzić czas na implementowanie własnego drzewa BST. Użycie hashset pozwoliło w drugim algorytmie umożliwiło skuteczne wykrywanie duplikatów przecięć w złożoności stałej.