Kacper Garus, 11.12.2023, ćwiczenie nr.4 Przecinanie odcinków

Dane techniczne:

Język - python, translator - Visual Studio Code, procesor - Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz, system operacyjny - Windows 10 Home 64bit

Realizacja ćwiczenia:

Ćwiczenie polegało na implementacji algorytmu sprawdzającego, czy dla danego zbioru odcinków istnieje choć jedno przecięcie i algorytmu, który dla danego zbioru odcinków zwraca: liczbę przecięć, listę współrzędnych przecięć oraz indeksy odcinków przecinających się.

W obu algorytmach strukturą stanu i strukturą zdarzeń jest SortedSet() z biblioteki sortedcontainers. SortedSet() przechowuje elementy w uporządkowanej kolejności według danego klucza dla obiektu. W strukturze zdarzeń przechowujemy obiekty , które reprezentują nasze punkty, uporządkowane według współrzędnej x, natomiast w strukturze stanu przechowujemy obiekty, które reprezentują odcinki, uporządkowane według wartości ich współrzędnej y dla danego punktu x, w którym znajduje się miotła. Dla obu algorytmów struktury są takie same, jednak w algorytmie sprawdzającym, czy istnieje przecięcie mogliśmy ją uprościć, bo nie ma potrzeby zamieniania miejscem w strukturze linii po wykryciu przecięcia.

Obsługa zdarzeń wygląda następująco: jeśli punkt jest początkiem odcinka to aktualizujemy współrzędną x punktu dla wszystkich linii, aby zachować porządek w strukturze stanu, gdy będziemy do niej wstawiać linię której początkiem jest ten punkt. Gdy wstawimy linię sprawdzamy czy przecina się z dolnym i górnym sąsiadem(jeśli istnieją). Jeśli punkt jest przecięciem to zamieniamy pozycje linii w strukturze stanu, za pomocą usunięcia ich z niej i wstawienia ponownie, po uwcześniejszym zaaktualizowaniu ich współrzędnej x dodając do niej małą wartość aby wartości y nie były sobie równe. Następnie dla wyższego w tym położeniu odcinka sprawdzam czy nie przecina się z górnym sąsiadem(jeśli istnieje), a dla niższego czy nie przecina się z dolnym sąsiadem(jeśli istnieje). Jeśli punkt jest końcem odcinka to sprawdzamy czy obaj sąsiedzi odcinka którego jest końcem(górny i dolny, jeśli istnieją) przecinają się ze sobą, a następnie usuwamy ten odcinek ze struktury stanu.

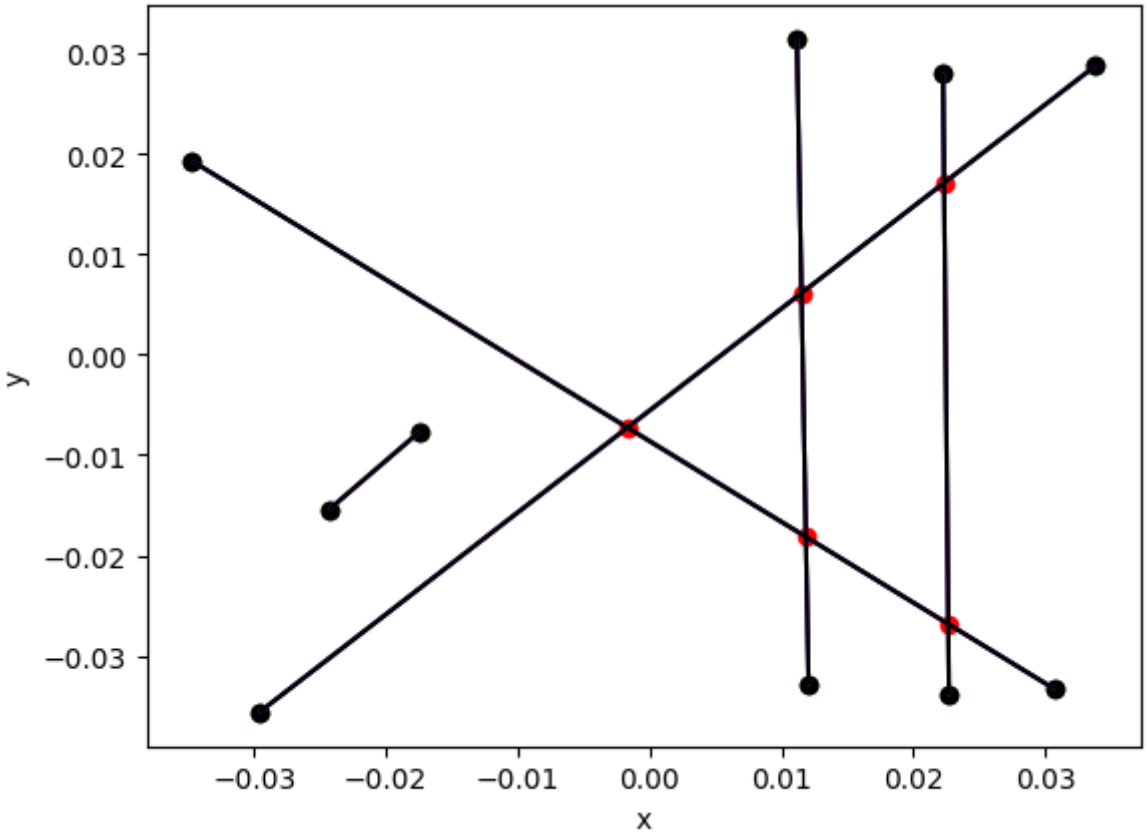
Sprawdzanie przecięć prostych odbywa się za pomocą wyznaczników oraz parametrycznej reprezentacji linii. Sprawdzamy czy przecięcie nie leży poza odcinkami, jeśli nie to liczymy punkt przecięcia. Następnie sprawdzamy, czy przecięcie tych dwóch linii nie zostało już wykryte(eliminowanie duplikatów), jeśli nie to punkt dodajemy do struktury zdarzeń i listy wynikowej.

Do algorytmów zaimplementowałem również ich wizualizacje które w formie gifów znajdują się w pliku z kodem, gdzie:

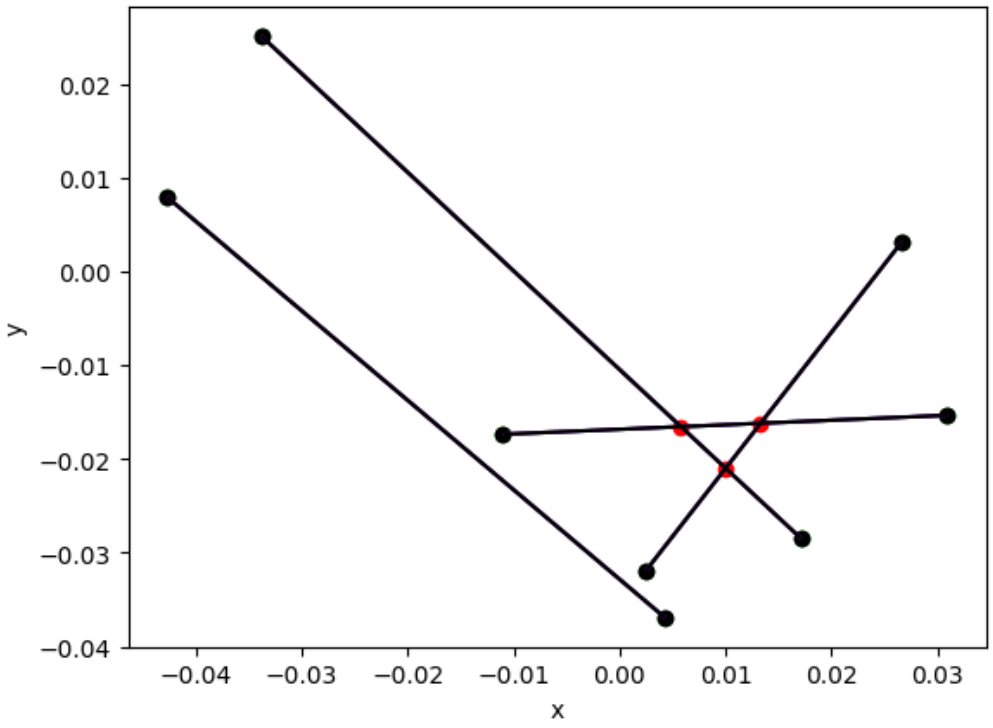
- Pionowa czerwona linia - wskazuje miejsce w którym znajduje się miotła
- Fioletowe linie - linie znajdujące się aktualnie w strukturze stanu
- Czarne linie - linie które zostały usunięte ze struktury stanu
- Czerwone punkty - punkty przecięć linii
- Zielone punkty – punkty w strukturze zdarzeń
- Czarne punkty – punkty, przez które przeszła miotła

Wyniki i analiza:

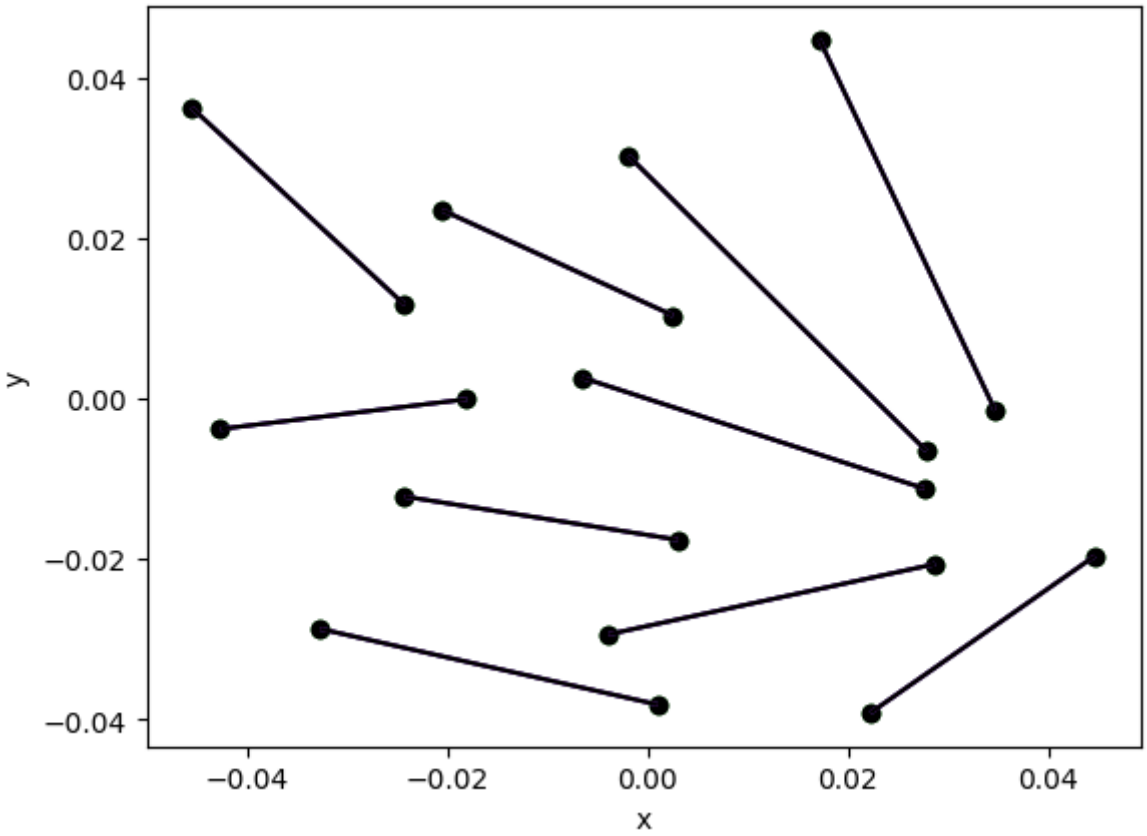
Działanie algorytmu na przykładowym zbiorze nr.1, w którym bez sprawdzania duplikatów pojawiły by się one:



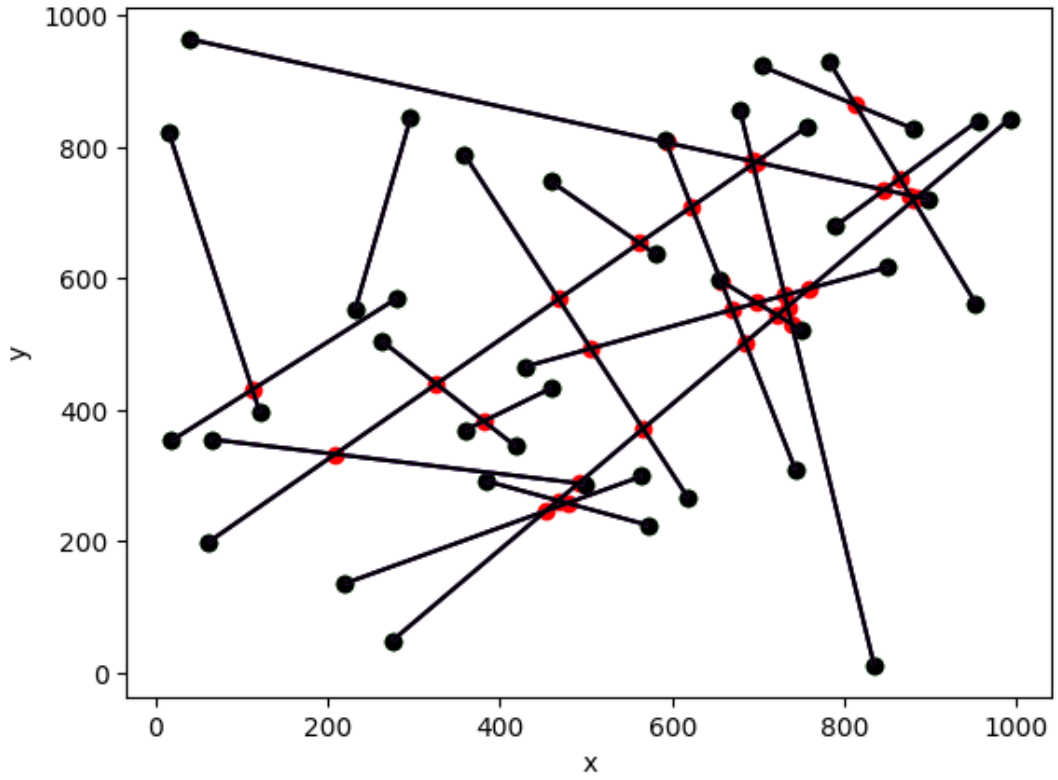
Działanie algorytmu na przykładowym zbiorze nr.2



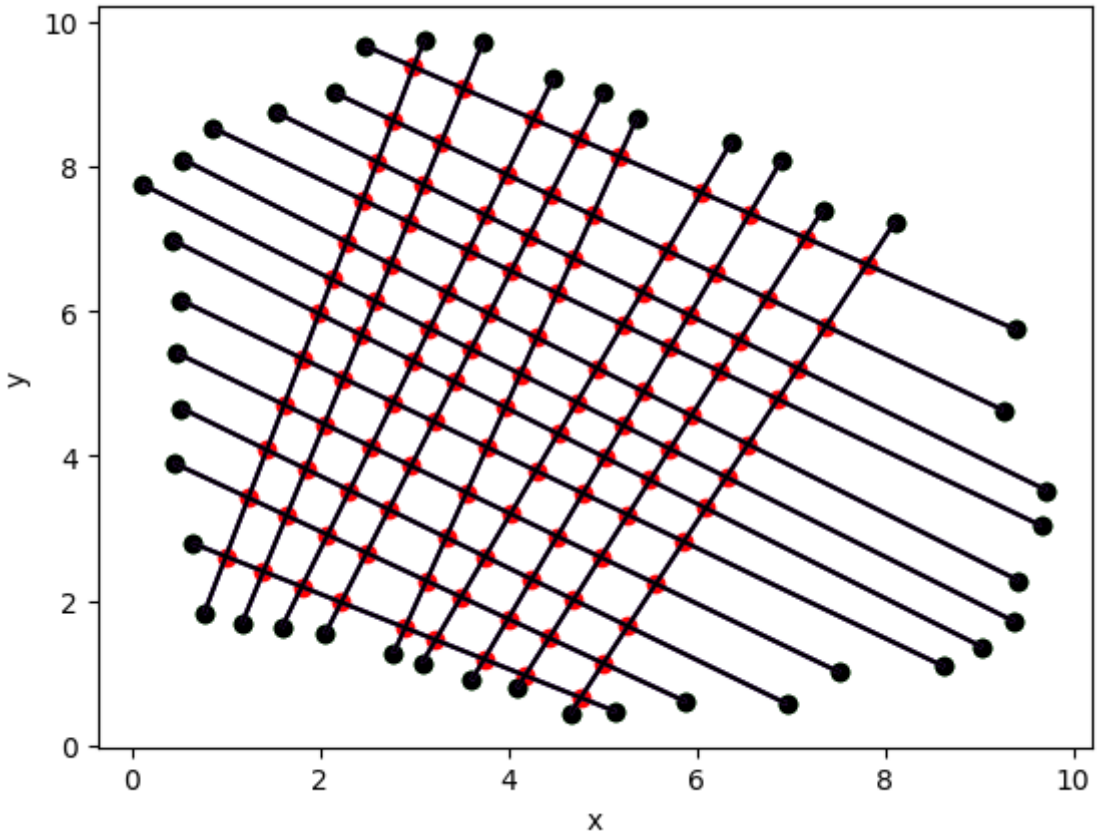
Działanie algorytmu na przykładowym zbiorze nr.3 w którym nie ma żadnych przecięć:



Działanie algorytmu na przykładowym zbiorze nr.4, który jest zbiorem 20 wygenerowanych losowo punktów (x, y) z zakresu $0 \leq x \leq 1000$, $0 \leq y \leq 1000$:



Działanie algorytmu na przykładowym zbiorze nr.5, który jest zbiorem wprowadzonym przeze mnie za pomocą myszki i układa się w siatkę:



Wnioski:

Oba algorytmy działają poprawnie na testowanych przeze mnie odcinkach patrząc na ich wizualizacje, oraz przechodzą testy dane od KN BiT. Dzięki strukturze SortedSet(), algorytmy działają sprawnie i optymalnie. W wynikach algorytmu nie występują duplikaty za sprawą sprawdzania, czy przecięcie danych odcinków nie pojawiło się już wcześniej.