Algorytmy Geometryczne
Sprawozdanie 4

Aga Patro
czw_13.30_A

1. Specyfikacja sprzętu i narzędzia wykorzystane w realizacji

System: Debian Linux Parrot OS x64
Procesor: AMD Ryzen 5 4500U, 6 rdzeni, 6 wątków, 4.00GHz
Pamięć RAM: 16 GB
Środowisko: Jupyter Notebook
Język: Python 3
Narzędzie pomocnicze: plik *geometria.ipynb* dostarczony z poleceniem, biblioteki *numPy*, *math, random*, *matplotlib* oraz *sortedcontainers*

2. Temat ćwiczenia

Celem ćwiczenia było zapoznanie się z tematem przecinania się odcinków na płaszczyźnie, oraz implementacja i przetestowanie algorytmu zamiatania. Algorytm zaimplementowałam w dwóch wersjach: sprawdzającej czy choć jedna para w zbiorze się przecina i w wersji znajdującej wszystkie punkty przecięcia.

3. Sposób wykonania

By wykonać ćwiczenie podjęłam następujące kroki:

1. Zaimplementowałam funkcję umożliwiającą interaktywne wprowadzenie odcinków jak i ich losowe generowanie.
2. Zaimplementowałam algorytm który sprawdza, czy w zbiorze odcinków choć jedna para się przecina.
3. Zaimplementowałam algorytm wyznaczający wszystkie punkty przecięcia w zbiorze.

4. Realizacja ćwiczenia

4.1 Algorytm sprawdzający czy w zbiorze choć jedna para się przecina

By zaimplementować algorytm musiałam zaimplementować dwie wymagane struktury danych:

1. Struktura zdarzeń - zrealizowana za pomocą posortowanej listy krotek które zawierają w sobie: (informację, czy punkt jest początkiem odcinka; współrzędna x dla której występuje zdarzenie; odcinek dla którego występuje zdarzenie). Elementy krotki sortujemy po współrzędnych x, czyli po drugim elemencie krotki.

2. Struktura stanu - zrealizowana za pomocą klasy SortedSet z biblioteki sortedcontainers. Jest to posortowany zmienny zestaw, w którym wartości są unikalne i posortowane w odpowiedniej kolejności. Klasa ta jest zrealizowana za pomocą drzewa czerwono-czarnego. Kolejność odcinków ustalana jest w następujący sposób: podczas dodawania/usuwania elementu ze struktury, porównuję odcinki ze względu na wartość

współrzędnej y odcinka dla x, w którym zdarzenie ma miejsce. Jest to możliwe dzięki klasie pomocniczej X_cord, która zawiera obiekt X, który zawsze przyjmuje wartość współrzędnej x obecnie obsługiwanego zdarzenia.

Dodatkowo zaimplementowałam strukturę Line która przechowuje w sobie id odcinka, oraz współczynniki tworzące wzór na prostą. Dodatkowo zawiera w sobie metody porównywania i zwracania hasha.

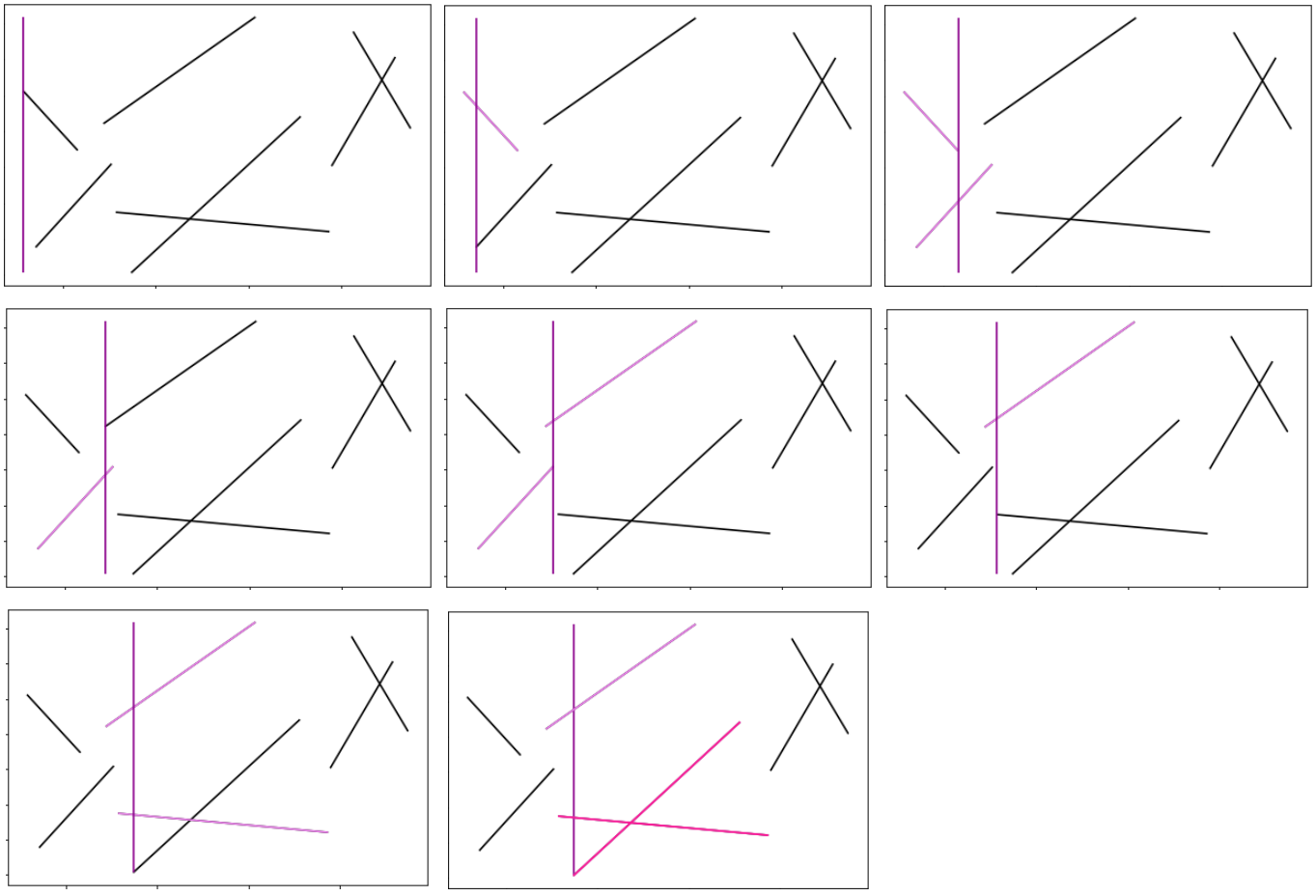
Algorytm polega na przetworzeniu listy odcinków na obiekty klasy Line, następnie szuka punktu przecięcia w oparciu o algorytm zamiatania: przegląda zdarzenia i sprawdza czy usunięcie/dodanie odcinka do struktury zdarzeń skutkuje znalezieniem przecięcia:

Jeśli zdarzeniem jest dodanie odcinka to sprawdzam czy zachodzi przecięcie między dodawanym odcinkiem a odcinkami które już znajdują się w strukturze stanu

Natomiast jeśli zdarzeniem jest usunięcie odcinka to sprawdzam czy zachodzi przecięcie pomiędzy odcinkami zlokalizowanymi w strukturze stanu nad i pod usuwanym odcinkiem

Algorytm zwraca krotkę ze stanem znalezienia punktu przecięcia (True/False), oraz sceny potrzebne do wizualizacji przebiegu algorytmu.

Poniżej możemy zobaczyć przebieg algorytmu:
kolor ciemny fiolet oznacza miotłę, kolor lawendowy oznacza odcinki które znajdują się w obecnej strukturze stanu miotły, kolor różowy oznacza odcinki które się przecinają



Zestaw 4.1.1 Przebieg algorytmu 1

4.2 Algorytm znajdujący wszystkie przecięcia w zbiorze

By zaimplementować algorytm musiałam zaimplementować dwie wymagane struktury danych:

1. Strukturę zdarzeń - klasa *event_struct*, którą realizuje za pomocą klasy SortedSet z biblioteki sortedcontainers. W tym przypadku zestaw sortujemy według współrzędnej x. Klasa ta jest realizowana za pomocą drzewa czerwono-czarnego, więc umożliwia szybkie wstawianie i usuwanie elementu, oraz znalezienie najmniejszego z nich. Klasa *event_struct* jest rozszerzoną klasą Line z punktu 4.1. *event_struct* zawiera w sobie informację o współrzędnej y zdarzenia oraz drugiej linii.

2. Strukturę stanu - klasa *state_struct*, którą również realizuje za pomocą klasy SortedSet z biblioteki sortedcontainers. Dodatkowo klasa *state_struct* zawiera w sobie metody dodawania, usuwania i wstawiania, które są wywoływane gdy wydarzy się odpowiednie zdarzenie. Ta klasa różni się od tej użytej w punkcie 4.1 tym, że za znalezienie przecięcia jest odpowiedzialna dana struktura, poprzez dodanie do struktury zdarzeń odpowiedniego zdarzenia, nie zaś ukończenia programu.

Dodatkowo używam strukturę Line, taką jak w punkcie 4.1. Zaimplementowałam także strukturę Line_event, która reprezentuje zdarzenie przecięcia, i jest wykorzystywana w wizualizacji.

Algorytm najpierw przetwarza odcinki z danej listy odcinków na obiekty klasy Line, następnie przegląda wszystkie zdarzenia ze struktury zdarzeń. Poszczególne zdarzenia obsługiwane są w następujący sposób:

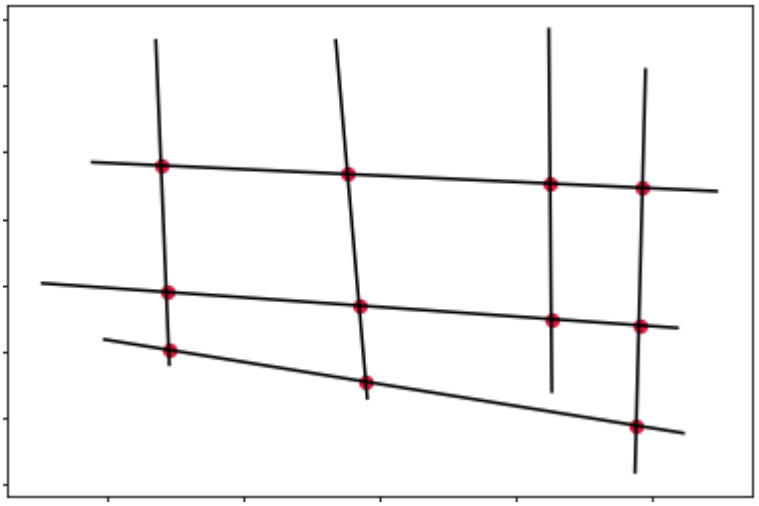
- Zdarzenie początku odcinka - wstawiam odcinek do struktury stanu i sprawdzam czy nie przecina on swojego poprzednika lub następnika w strukturze. Jeśli przecina, to dodaje zdarzenie przecięcia do struktury zdarzeń.
- Zdarzenie końca odcinka - usuwam odcinek ze struktury stanu i sprawdzam czy odcinki rozdzielane przez usunięty odcinek się przecinają. Jeśli się przecinają, to dodaje zdarzenie przecięcia do struktury zdarzeń.
- Zdarzenie przecięcia odcinków:
    - usuwam odcinek - A znajdujący się wyżej (w miejscu bezpośrednio przed przecięciem) w strukturze stanu,
    - sprawdzam czy niższy odcinek B przecina odcinek od którego był oddzielony przez odcinek A
    - usuwam odcinek B ze struktury
    - wstawiam do struktury odcinek A i sprawdzam, czy przecina odcinek który przed rozpoczęciem procesu znajdował się bezpośrednio pod odcinkiem B
    - wstawiam odcinek B do struktury, tak by w strukturze stanu znajdował się nad odcinkiem A. Robię to poprzez zwiększenie o wartość epsilon wartości X_cord, tak by kolejność odcinków została zmieniona, a na odcinku (x, x+epsilon) nie było żadnych zdarzeń.

Algorytm zwraca krotkę zawierającą zbiór zdarzeń przecięcia oraz sceny które umożliwiają wizualizację przebiegu algorytmu.

5. Testowanie obu algorytmów dla różnych zbiorów danych

5.1 Zbiór 1

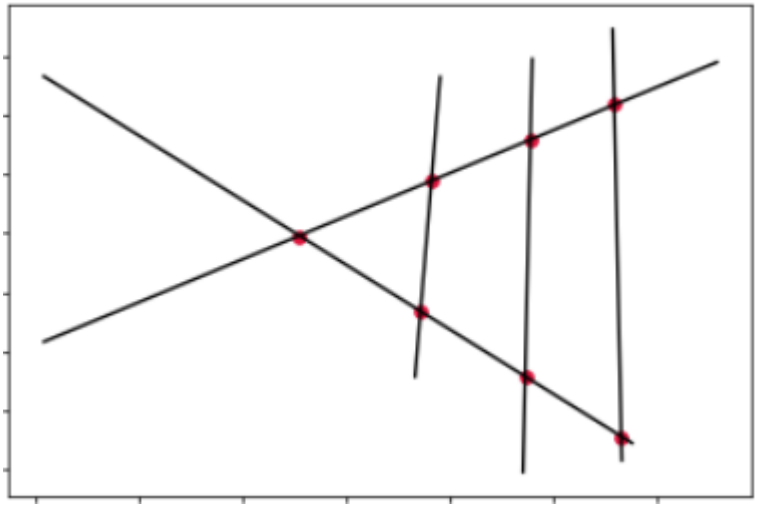
Algorytm 1 zwrócił True, natomiast algorytm drugi zwrócił liczbę przecięć wynoszącą 11, która jest zgodna z rzeczywistą



Rysunek 5.1.1 Wynik algorytmu 2 dla pierwszego zbioru danych

5.2 Zbiór 2

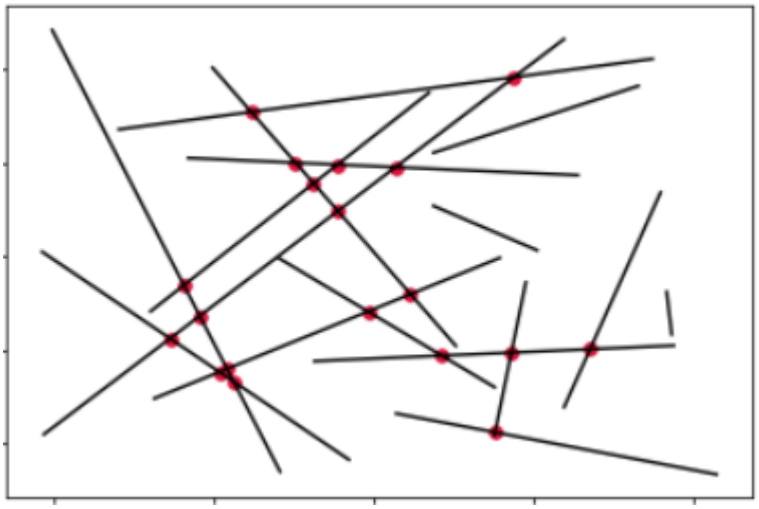
Algorytm 1 zwrócił True, natomiast algorytm drugi zwrócił liczbę przecięć wynoszącą 7, która jest zgodna z rzeczywistą.



Rysunek 5.1.2 Wynik algorytmu 2 dla drugiego zbioru danych

5.3 Zbiór 3

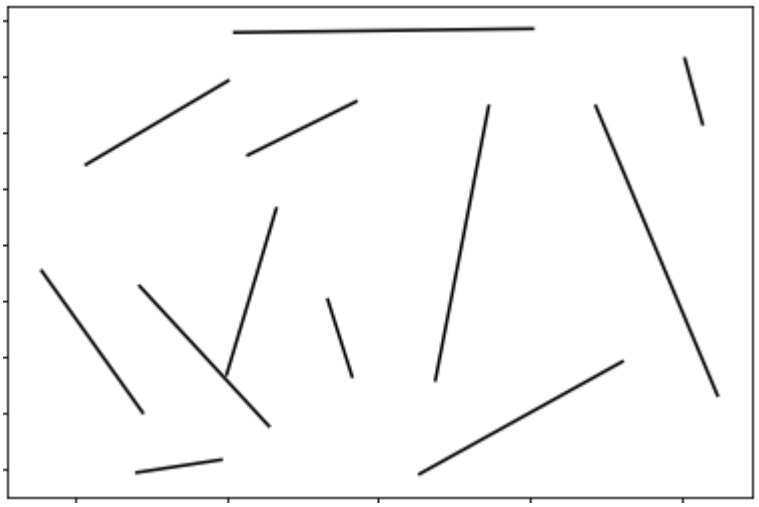
Algorytm 1 zwrócił True, natomiast algorytm drugi zwrócił liczbę przecięć wynoszącą 19, która jest zgodna z rzeczywistą.



Rysunek 5.1.3 Wynik algorytmu 2 dla trzeciego zbioru danych

5.4 Zbiór 4

Algorytm 1 zwrócił False, natomiast algorytm drugi zwrócił liczbę przecięć wynoszącą 0, która jest zgodna z rzeczywistą.



Rysunek 5.1.4 Wynik algorytmu 2 dla czwartego zbioru danych