
Algorytmy Geometryczne

Ćwiczenie 4 - Przecinanie odcinków

Jakub Własiewicz - Grupa 2 - Poniedziałek 13:00

2025-12-03

1. Dane techniczne

Program został uruchomiony na komputerze z następującymi specyfikacjami:

- **System Operacyjny** - Fedora Linux 43
- **Architektura Procesora** - x86_64
- **Procesor** - AMD Ryzen 7 7840HS
- **Język** - Python 3.14.0

Ćwiczenie realizowane było w środowisku *Jupyter*, do wizualizacji zostało użyte narzędzie stworzone przez koło naukowe *BIT* oraz następujące biblioteki:

- **matplotlib**
- **numpy**
- **pandas**
- **sortedcontainers**

Do obliczeń została użyta tolerancja dla zera $\varepsilon = 10^{-9}$, oraz liczby zmienno-przecinkowe o rozmiarze 64 bitów.

2. Opis ćwiczenia

Ćwiczenie polegało na implementacji algorytmu zamiatania w celu wyznaczenia przecięć odcinków na płaszczyźnie. Obejmowało to sprawdzenie istnienia przecięcia oraz wyznaczenia wszystkich punktów przecięć.

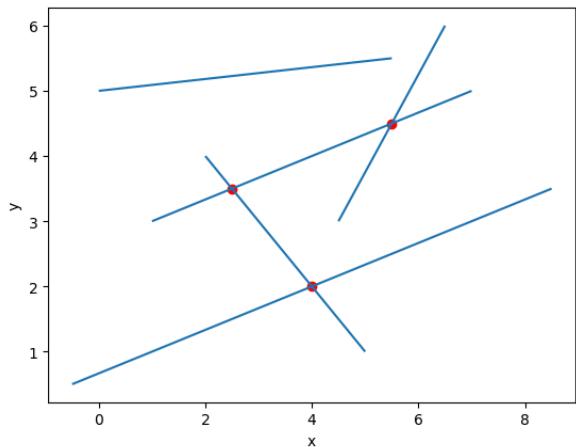
3. Wstęp teoretyczny

3.1. Przecięcie odcinków

Dla zbioru odcinków $S = \{s_1, s_2, \dots, s_n\}$ w \mathbb{R}^2 , przecięciem nazywamy taką parę (s_i, s_j) , że $i \neq j$ oraz $s_i \cap s_j \neq \emptyset$

Wprowadzimy następujące założenia:

- Żaden z odcinków nie jest pionowy,
- Końcowe współrzędne x -owe nie mogą się powtarzać dla każdej pary odcinków,
- Żadne trzy odcinki nie przecinają się w jednym punkcie,
- Para odcinków przecina się w co najwyżej jednym punkcie.



Rysunek 1: Przykładowy zbiór odcinków w \mathbb{R}^2 , z zaznaczonymi punktami przecięć

3.2. Algorytm zamiatania

Algorytm zamiatania polega na ustaleniu pewnej hiperplaszczyzny, w naszym przypadku będzie to prosta, którą będziemy przesuwać po osi x . Nazywamy tę prostą „miotłą”. Będzie się ona zatrzymywać w 3 różnych interesujących nas zdarzeniach: początek odcinka, koniec odcinka oraz punkt przecięcia. Pozycje te przetrzymujemy w strukturze zdarzeń Q , natomiast w strukturze stanu T przechowujemy uporządkowane względem współrzędnej y przecięcia odcinków z miotłą. Sprawdzane względem przecięcia będą tylko odcinki sąsiadujące ze sobą w strukturze stanu, czyli pierwsze odcinki nad oraz pod punktem przecięcia z miotłą.

4. Realizacja ćwiczenia

4.1. Wybrane struktury danych

Do realizacji algorytmu zamiatania, jako strukturę stanu T wykorzystano *SortedSet* z biblioteki *sortedcontainers*. Zapewnia ona łatwe porządkowanie odcinków oraz operacje dodawania, usuwania, wyszukiwania w czasie $O(\log n)$. Dzięki temu jesteśmy w stanie efektywnie sprawdzać czy sąsiednie odcinki się przecinają.

W przypadku struktury zdarzeń Q , do algorytmu weryfikacji istnienia przecięcia wykorzystana została lista początków i końców odcinków posortowana malejąco, aby wykorzystać operację *.pop()*, co pozwala uniknąć przesuwania pozostałych elementów w pamięci. Takie rozwiązanie jest wystarczające ze względu na zakończenie algorytmu w przypadku wykrycia przecięcia, co oznacza brak konieczności dodawania punktów przecięć do struktury zdarzeń.

Natomiast w algorytmie wyznaczania przecięć konieczna była zmiana struktury danych na kopiec, z powodu potrzeby dodawania punktu przecięcia oraz dostępu do najmniejszej współrzędnej x w czasie $O(\log n)$.

Dla prostego użytku zostały zaimplementowane klasy *Point* oraz *Section* oznaczające odpowiednio punkt oraz odcinek. Mają one zdefiniowane operatory porównywania na potrzebę działania struktury stanu i zdarzeń.

4.2. Sprawdzanie przecięcia dwóch odcinków

Aby sprawdzić czy dwa odcinki się przecinają stworzona została funkcja *check_intersections* zwracająca punkt przecięcia lub *None*, jeżeli takiego nie ma. W funkcji porównywane są współczynniki prostych, jeżeli są równe to punkt przecięcia nie istnieje albo jest ich nieskończonie wiele, a taki przypadek wykluczyliśmy w założeniach. Następnie, wyznaczona jest współrzędna *x* punktu przecięcia za pomocą układu równań i weryfikowana czy zawiera się w zakresach obu odcinków. Jeżeli jest to spełnione, współrzędną *y* otrzymujemy za pomocą równania jednej z prostych, zwracamy krotkę (x, y) .

4.3. Implementacja algorytmu wyznaczającego punkty przecięcia

Algorytm najpierw tworzy kopiec i umieszcza na nim wszystkie początki i końce odcinków, rozróżniając czy to lewy czy prawy koniec oraz ponumerowaną listę odcinków, aby zapewnić bezproblemowy dostęp do każdego z nich. Następnie ściągamy punkty z kopca dopóki nie jest pusty. Dla wyjętego punktu, przypisujemy jego współczynnik *x* do zmiennej statycznej, oznacza ona położenie miotły; jest potrzebna do porównywania odcinków. Potem zczytujemy odcinek do którego należy z poprzednio utworzonej listy. Po tym następuje rozgałęzienie ze względu na typ zdarzenia:

1. Punkt jest lewym końcem odcinka:

Dodajemy odcinek do struktury stanu *T* oraz aktualizujemy statyczną zmienną położenia miotły. Następnie przetwarzamy sąsiadów odcinka, sprawdzając czy występują punkty przecięcia, jeśli tak, dodajemy je na kopiec.

2. Punkt jest prawym końcem odcinka:

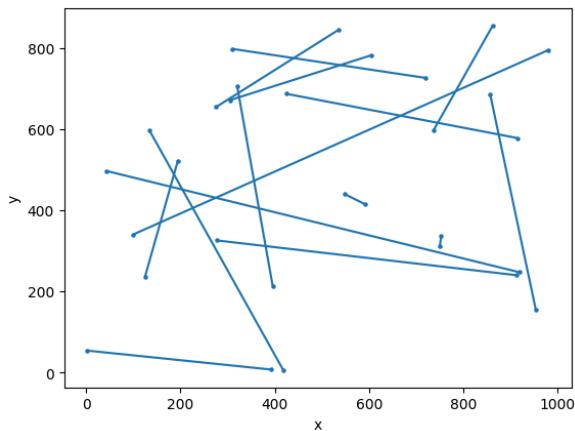
Aktualizujemy położenie miotły, sprawdzamy przecięcie między sąsiadami tego odcinka, jeśli istnieją. Po czym usuwamy odcinek z struktury stanu.

3. Punkt jest przecięciem:

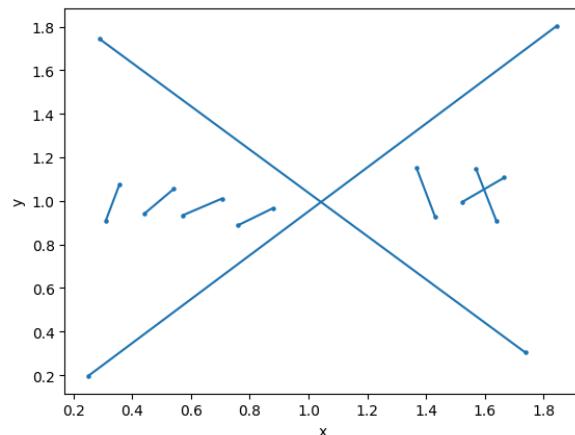
Zamieniamy kolejność odcinków których dotyczy przecięcie w strukturze stanu. Ponieważ w punkcie przecięcia jest to niejednoznaczne, to ustawiamy położenie miotły na $x + \varepsilon$, gdzie $\varepsilon = 10^{-9}$. Przetwarzamy sąsiadów tych odcinków po zamianie.

Ponieważ jest możliwość, że przecięcie zostanie przetworzone więcej niż jeden raz, używany jest zbiór który pozwala sprawdzić czy dane przecięcie już nastąpiło.

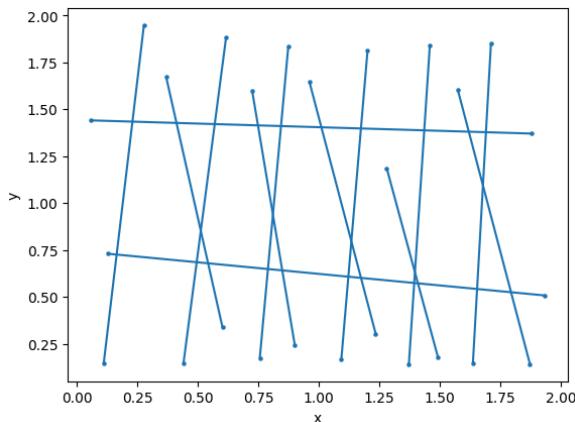
4.4. Wybrane zbiory testowe



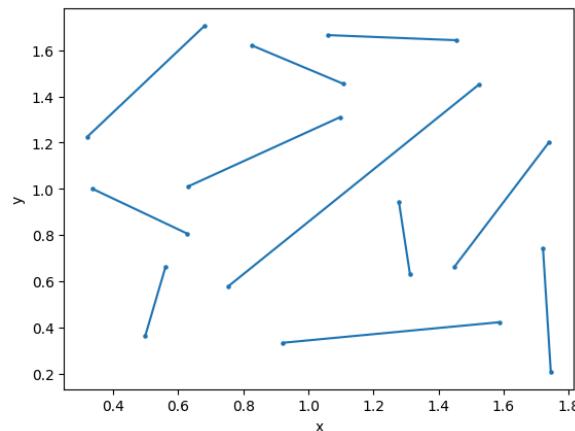
Rysunek 2: Zbiór A: 15 losowo wygenerowanych odcinków o punktach ze współrzędnymi z zakresu $[-1000, 1000]$



Rysunek 3: Zbiór B: Dwa przecinające się odcinki, oraz mniejsze ustawione pomiędzy nimi



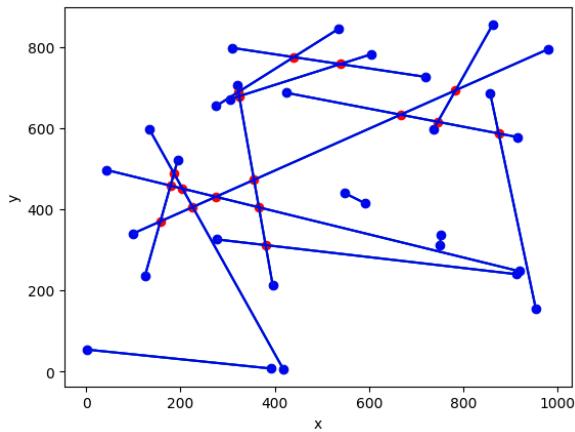
Rysunek 4: Zbiór C: 2 odcinki przecinające pozostałe odcinki



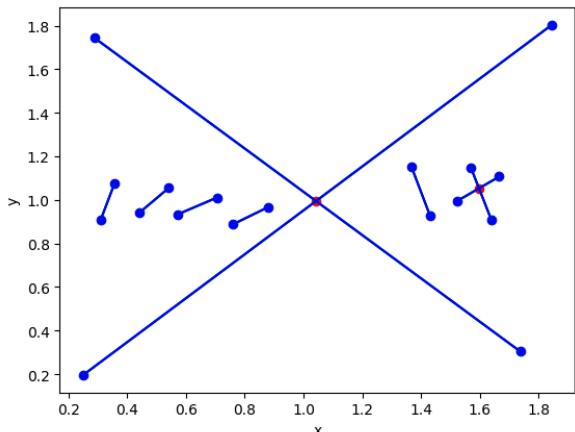
Rysunek 5: Zbiór D: 11 nieprzecinających się odcinków

5. Analiza wyników

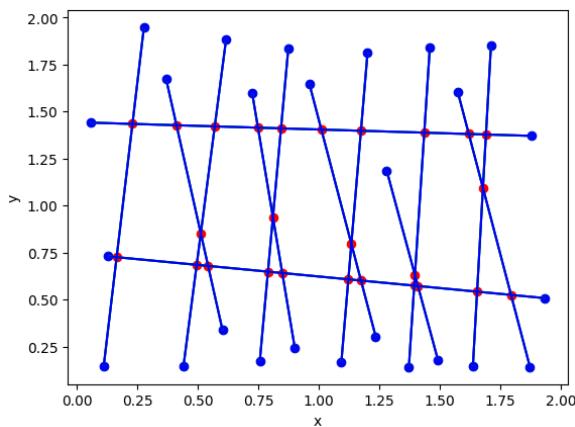
Poniżej zestawiono wynik wizualizacji przecięć punktów odcinków w wybranych zbiorach, punkty te są zaznaczone kolorem czerwonym.



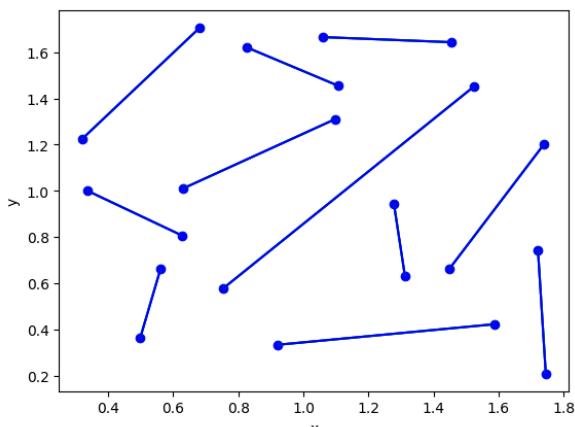
Rysunek 6: Punkty przecięć dla zbioru A



Rysunek 7: Punkty przecięć dla zbioru B



Rysunek 8: Punkty przecięć dla zbioru C



Rysunek 9: Punkty przecięć dla zbioru D

Zbiór	Liczba punktów przecięć
A	17
B	2
C	26
D	0

Tabela 1: Liczba punktów przecięć wyznaczonych dla zbiorów testowych

Algorytm poprawnie poradził sobie ze zbiorami testowymi. Zbiór A testował algorytm dla losowych danych. Dla zbioru B przewidujemy podwójne zliczenie przecięcia w przypadku niepoprawnego algorytmu, w tabeli 1 mamy dowód poprawności zliczania punktów przecięć dla zbioru B. Zbiór C testował przypadek wielu przecięć. Dla zbioru D poprawnie nie został zakwalifikowany żaden punkt jako przecięcie odcinków.

6. Wnioski

Algorytm zamiatania poprawnie zadziałał dla wszystkich zbiorów testowych. Ważnym okazał się wybór struktury zarówno zdarzeń jak i stanu. Dzięki złożoności czasowej tych struktur i metodzie zamiatania, jesteśmy w stanie osiągnąć lepszą złożoność od $O(n^2)$, zależącą od liczby przecięć.

Animacje algorytmu w postaci plików *.gif* zostały załączone w archiwum dołączonym ze sprawozdaniem.