Algorytmy geometryczne
laboratorium 4 - sprawozdanie

Radosław Rolka
Informatyka WI, II rok
Gr 6, tydz. A czwartek 13:00

Spis treści

1 Wstęp

1.1 Cel ćwiczenia

Ćwiczenie wprowadzające w zagadnienia wykrywania przecięć odcinków – implementacja algorytmów sprawdzających, czy zadane odcinki się przecianają, znalezienie współrzędnych tych punktów oraz identyfikacja tych odcinków. Ponadto przeprowadzenie testów, wizualizacja i opracowanie wyników.

1.2 Program ćwiczenia

- Zaimplementuj funkcję generującą losowo zadaną liczbę odcinków z podanego zakresu współrzędnych 2D
- Zaimplementuj interaktywne wprowadzanie odcinków przez użytkownika za pomocą myszki
- Zaimplementuj algorytm sprawdzający, czy zadane odcinki się przecianają
- Zaimplementuj algorytm, który znajduje wszystkie przecięcia na zbiorze zadanych odcinków oraz podanie odcinków do których należą i współrzędnych przecięć

2 Wykorzystane narzędzia

2.1 Środowisko

Ćwiczenie zostało wykonane w Jupyter Notebook wykorzystując język programowania Python oraz dodatkowe biblioteki, które zostały zawarte w projekcie dostarczonym na zajęciach.

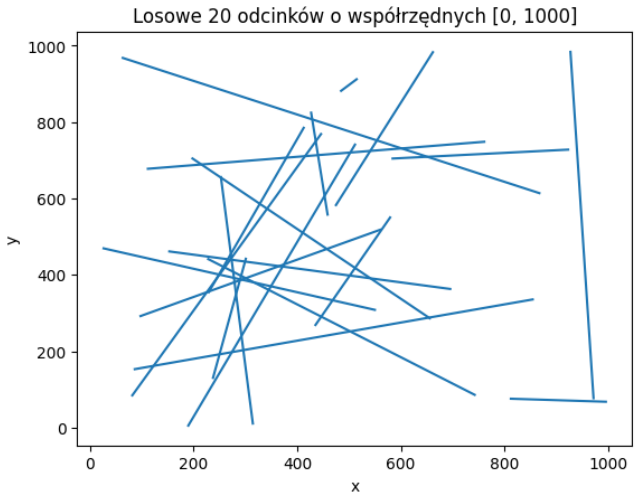
2.2 Sprzęt

Do wykonania został wykorzystany procesor Intel(R) Core(TM) I5-10300H 2.50GHz oraz system operacyjny Microsoft Windows 10 64bit ver 22H2.

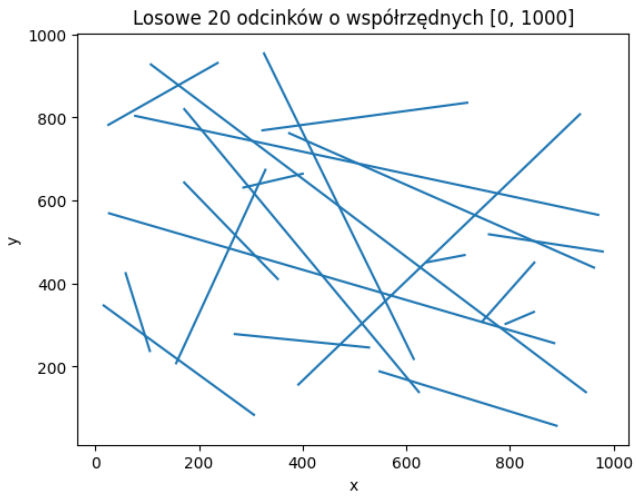
3 Przebieg ćwiczeń

3.1 Generowanie losowych odcinków na płaszczyźnie

Do wygenerowania losowych odcinków na płaszczyźnie wykorzystałem funkcję np.random.uniform, która zapewnia pseudolosowe generowanie unikalnych liczb. Ponadto żaden odcinek nie jest pionowy, ani żadna para odcinków nie ma końców o tej samej współrzędnej x.



Rysunek 1: Przykład wygenerowanych odcinków



Rysunek 2: Przykład wygenerowanych odcinków

3.2 Wykrywanie istnienia przecięć między odcinkami

Algorytm wykrywający, czy dowolne dwa odcinki się przecinają implementuje strukturą zwaną "miotłą". W trakcie zamiatania idziemy po kolejnych punktach krytycznych, posortowanych rosnąco względem współrzędnej x. Te punkty krytyczne, czyli początki i końce odcinków, są przechowywane w kolejce priorytetowej *queue.PriorityQueue*, nazywanej później Q. W punktach krytycznych aktualizujemy kolejność odcinków względem ich współrzędnej y w danym x. Implementując skorzystałem ze struktury *sortedcontainers.SortedSet*, która implementuje drzewo czerwono-czarne i będzie nazywana T. W operacji wstawienia/usunięcia odcinka ze struktury T sprawdzamy parametrycznie, czy odcinki z nowymi sąsiadami się przecinają. Operacje powtarzamy do momentu wykrycia przecięcia, zwracając wartość $True$, lub do momentu przejścia przez wszystkie punkty krytyczne znajdujące się w T i zwracając wartość $False$.

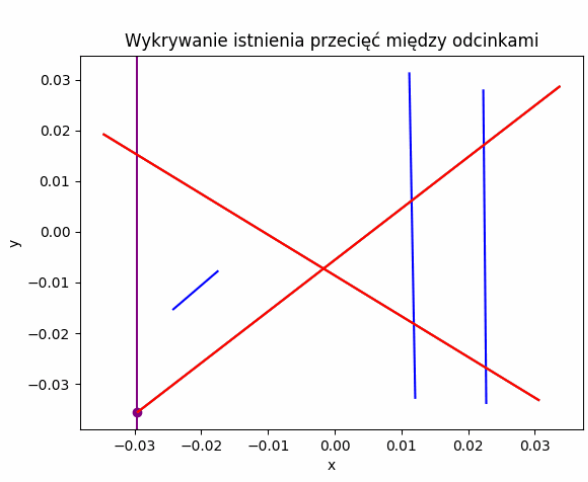
3.2.1 Złożoność

- Inicjalizowanie listy punktów krytycznych Q - $O(n \log n)$
- Aktualizowanie struktury T - $O(n \log n)$
- **Złożoność całkowita** - $O(n \log n)$

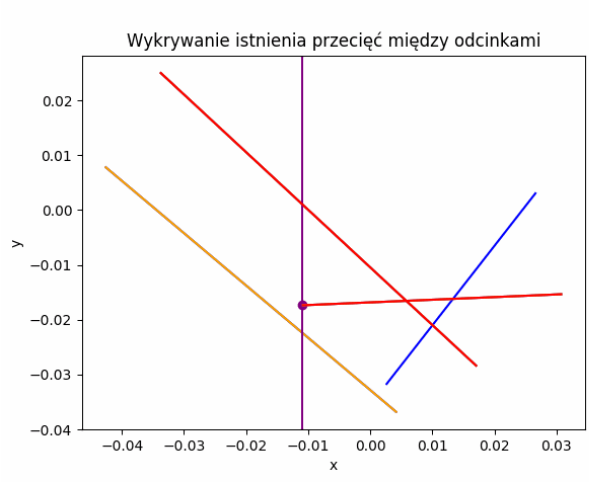
3.2.2 Przykłady

Legenda:

- **Odcinek bazowy**: Poza przetwarzaniem, nie znajduje się w T
- **Odcinek przetwarzany**: Miotła znajduje się pomiędzy jego początkiem i startem, jest możliwe wykrycie przecięcia
- **Miotła**: Prosta wskazująca aktualne położenia przetwarzanego punktu krytycznego (punkt został dodatkowo zaznaczony).
- **Odcinki przecinające się**: Na końcu odcinki zaznaczone na czerwono to odcinki przecinające się (jeśli takie istnieją).



Rysunek 3: Przykład wykrytego przecięcia



Rysunek 4: Przykład wykrytego przecięcia

3.3 Detekcja wszystkich przecięć między odcinkami

Algorytm wyznaczający wszystkie punkty przecięć między danymi odcinkami bazuje na idei poprzedniego algorytmu. Jedynie różnice dotyczą zachowania w trakcie wykrycia przecięcia, mianowicie zamiast zakończenia pracy programu, punkt przecięcia zostaje dodany do struktury Q (jeśli nie jest początkiem/końcem odcinka), a w trakcie późniejszego jego przetwarzania aktualizujemy strukturę T zamieniając krzyżujące się odcinki kolejnością i sprawdzamy możliwe przecięcia z nowymi sąsiadami (pomijamy ponowne sprawdzenie zamienionych odcinków). Powyższe instrukcje są wykonywane dla wszystkich wykrytych przecięć, natomiast reszta metod działa bez zmian.

Aby uniknąć wielokrotnego dodawania tego samego punktu przcięcia do struktury Q, przechowuję krotki przecięć (indeksy odcinków) w zbiorze set, który zapewnia aktualizacje zawartości oraz wyszukiwanie elementów w czasie stałym.

3.3.1 Złożoność

- Inicjalizowanie listy punktów krytycznych Q - $O(n \log n)$
- Aktualizowanie struktury T - $O((P+n) \log n)$
- Aktualizowanie struktury Q - $O(P \log n)$
- **Złożoność całkowita** - $O((P+n) \log n)$

Gdzie P to liczba przecięć.

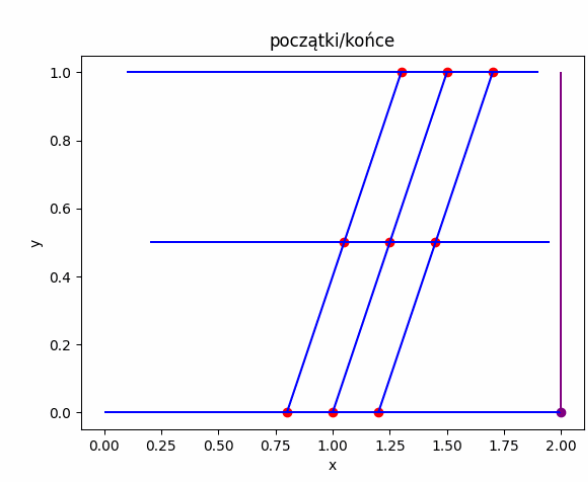
3.3.2 Wymagające przykłady

- Przecięcie, którym jest jeden z punktów końcowych odcinka
- Kratka/Posadzka
- Wielokrotne wykrycie jednego przecięcia

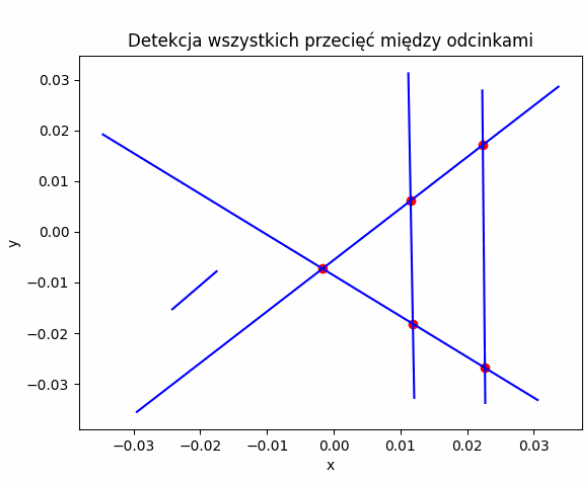
3.3.3 Przyklady

Legenda:

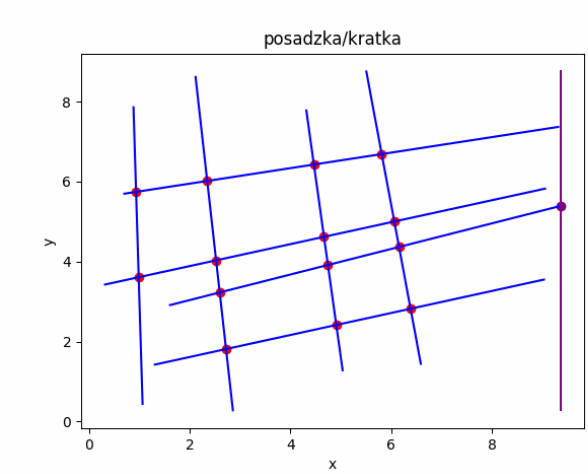
- **Odcinek bazowy**: Poza przetwarzaniem
- **Odcinek przetwarzany**: Miotła znajduje się pomiędzy jego początkiem i startem, jest możliwe wykrycie przecięcia
- **Miotła**: Prosta wskazująca aktualne położenia przetwarzanego punktu krytycznego (punkt został dodatkowo zaznaczony).
- **Punkty przecięcia**: Na końcu punkty zaznaczone na czerwono to punkty przecięć (jeśli takie istnieją).



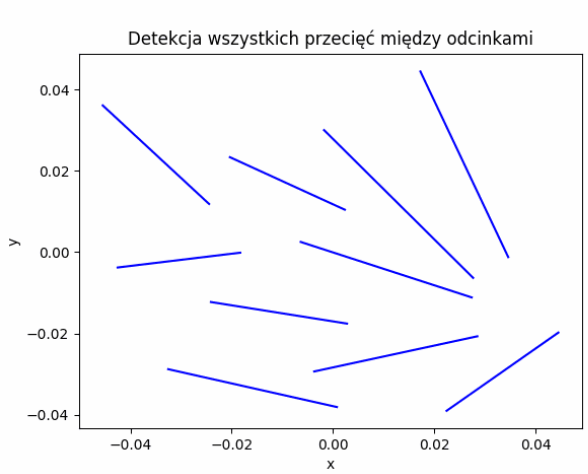
Rysunek 5: Ukończone detekcje wszystkich przecięć



Rysunek 6: Ukończone detekcje wszystkich przecięć



Rysunek 7: Ukończone detekcje wszystkich przecięć



Rysunek 8: Ukończone detekcje wszystkich przecięć

4 Wnioski

Wszystkie algorytmy przeszły poprawnie wszystkie testy. Ponadto w trakcie opracowywania wyników nie zauważyłem żadnych błędów w działaniu. Można zatem stwierdzić, że ich implementacja nie zawiera żadnych błędów, a ich działanie spełnia postawione wymagania.