Opracowanie wyników laboratoriów

Środowisko

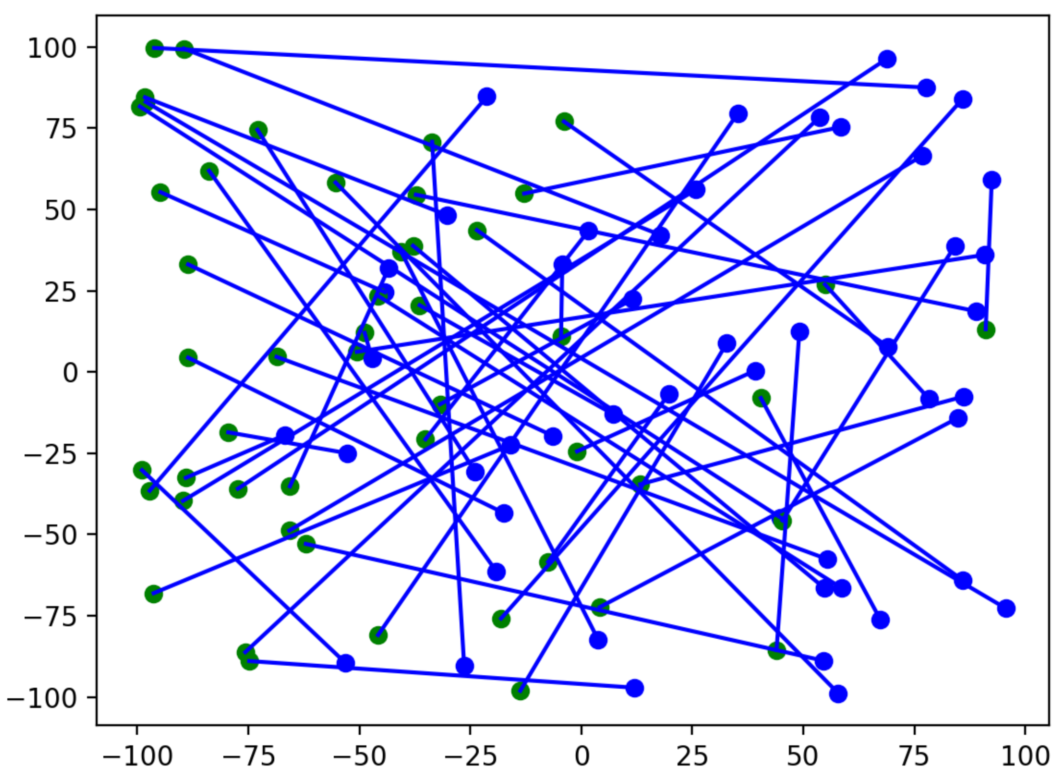
Wszystkie operacje wykonane zostały na komputerze stacjonarnym z procesorem i5-7600k, RAM 32GB. Językiem z jakiego korzystałem był Python w wersji 3.10. Program napisany został w *Jupyter Notebook*.

Generowanie zbioru odcinków

W programie zaimplementowałem funkcje generującą zbiór n odcinków, w których obie współrzędne każdego końca odcinka spełniają:

$$a < x < b \ \wedge \ a < y < b, \tag{1}$$

gdzie a i b to stałe przekazywane funkcji. Ponadto funkcja zapewnia, że żaden z odcinków nie jest wertykalny oraz żadne dwa punkty tworzące odcinki nie posiadają takich samych współrzędnych x-owych. Punkty generowane są za pomocą rozkładu jednostajnego. Przykładowe wywołanie funkcji to `genSegments([-100, 100], [-100, 100], 50)`, co wygeneruje następujący zbiór odcinków. Kolorem zielonym oznaczone są punkty początkowe patrząc od lewej, natomiast niebieskim odcinki wraz z końcami.



Wizualizacja 1: Wygenerowane odcinki.

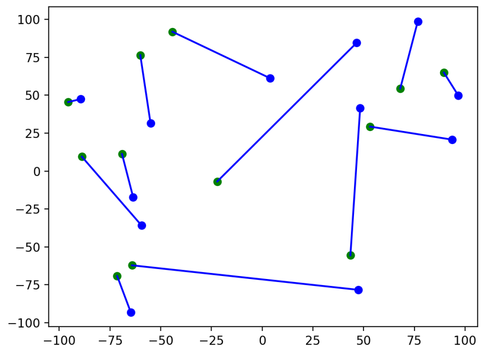
Sprawdzenie czy w zbiorze odcinków chociaż dwa się przecinają

Możemy to uzyskać za pomocą algorytmu Shamosa-Hoeya. Jest to algorytm opierający się głownie na instancji "miotły", która zawiera odcinki podejrzane o przecięcia. Pozwala to dla zbioru zawierającego n odcinków, stwierdzić, czy istnieją przynajmniej dwa, które się przecinają w czasie $O(n \log n)$, co jest znacznym usprawnieniem w porównaniu do algorytmu sprawdzającego każdą parę odcinków z złożonością $O(n^2)$. Możemy to osiągnąć korzystając z struktury zdarzeń będącego listą zawierającą `[idx, [p.x, p.y]]`, gdzie `idx` jest indeksem danego segmentu w zbiorze wszystkich segmentów, natomiast `p.x` i `p.y` to odpowiednio współrzędna x-owa i y-owa punktu. Zauważmy, że struktura ta posiadać będzie $2n$ elementów, jako, że każdy z n odcinków zawiera początek i koniec. Posortowanie takiej struktury po współrzędnej x-owej to $2n \log(2n)$, co daje złożoność $O(n \log n)$. Następnie utworzyć możemy strukturę stanu, która powinna zachowywać porządek elementów oraz wspierać dodawanie i usuwanie w czasie $O(\log n)$. Zauważmy, że posortowana lista w pythonie posiada te właściwości. Korzystając z binarnego wyszukiwania możemy znaleźć odpowiedni element w liście oraz w zamortyzowanym czasie $O(1)$ możemy go usuwać lub dodawać. Takie wyszukiwanie znajduje się w bibliotece standardowej **bisect**. Możemy w strukturze stanu przechowywać trójki `[idx, a, b]`, gdzie `idx` jest indeksem danego segmentu w zbiorze wszystkich segmentów, natomiast `a` i `b` to odpowiednio współczynnik kierunkowy i wyraz wolny prostej przechodzącej przez dany odcinek. Dzięki temu oszczędzamy na częstym korzystaniu z drogiej operacji dzielenia. Kolejne kroki można opisać jako powtórzenie $2n$ razy następujących kroków:

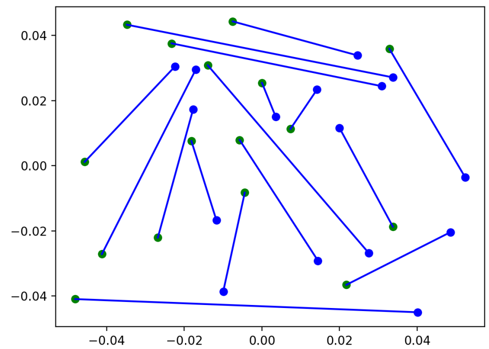
1. Jeżeli trafiliśmy na odcinek, którego nie ma w "miotle", to korzystając z binarnego wyszukiwania szukamy jego miejsca w strukturze stanu relatywnie do współrzędnych y-owych punktów zawartych w miotle dla x równego współrzędnej dodawanego punktu i dodajemy go. Następnie sprawdzamy przecięcie z sąsiednimi odcinkami w "miotle", jeżeli takowe istnieją. Są to operacje z pesymistycznymi złożonościami $O(\log n) + O(\text{const}) + O(\text{const}) = O(\log n)$.

2. Jeżeli trafiliśmy na odcinek zawierający się w "miotle", znajdujemy go i usuwamy w $O(\log n)$ oraz sprawdzamy, czy jego sąsiedzi się przecinają, o ile istnieją.

Zauważmy, że każda z operacji jest wykonywana $2n$ razy co daje $O(n \log n)$. Funkcje tą zaimplementowałem jako `intersectionExists(segments)`, gdzie `segments` jest listą zawierającą odcinki w postaci `[[p1.x, p1.y], [p2.x, p2.y]]`. Zwraca ona `True`, jeżeli wykryje przecięcie. Poprawność możemy przetestować poprzez generowanie danych, aż nie znajdziemy takiego zbioru, w którym nie ma przecięć. Poniżej w *Wizualizacji 2* zamieszczam dwa zbiory punktów, dla których funkcja nie wykrywa przecięć oraz w *Wizualizacji 3* dwa dla których wykrywa.

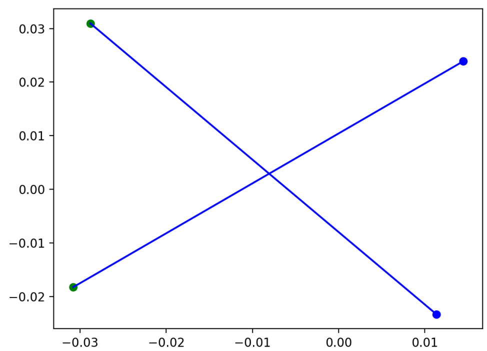


(a) Zbiór 1.

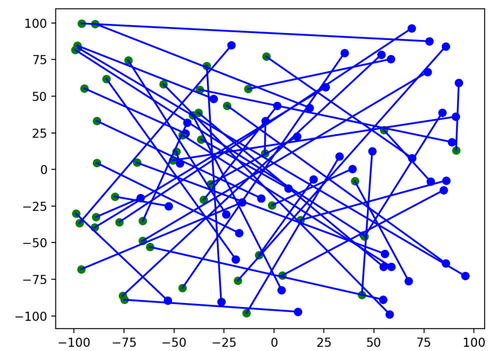


(b) Zbiór 2.

Wizualizacja 2: Zbiory niezawierające przecięć.



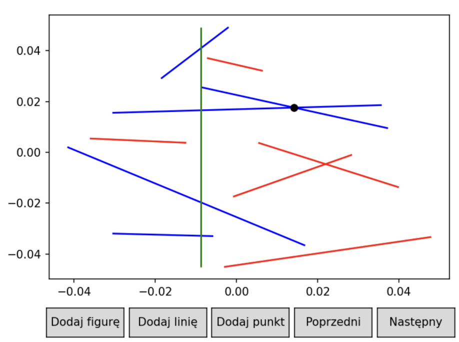
(c) Zbiór 3.



(d) Zbiór 4.

Wizualizacja 3: Zbiory zawierające przecięcia.

W *Wizualizacji 4* poniżej zamieszczam, także moment znalezienia punktu przecięcia w wizualizacji działania algorytmu. Kolorem czerwonym zaznaczone są punkty nienależące do miotły, niebieskim te należące, zielono reprezentacja miotły, natomiast na czarno widzimy znaleziony punkt przecięcia.



Wizualizacja 4: Graficzna reprezentacja znalezienia punktu.

Wykrywanie wszystkich przecięć

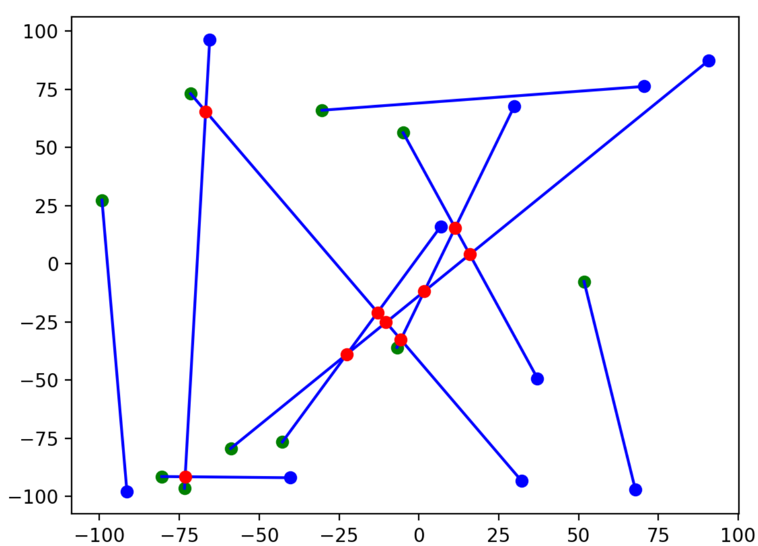
Aby znaleźć wszystkie przecięcia możemy skorzystać z algorytmu Bentleya-Ottmanna, który działa na bardzo podobnej zasadzie do algorytmu Shamosa-Hoeya. Analogicznie do tego poprzedniego jest on oparty na strukturze "miotły", jednak do struktury zdarzeń dodajemy także punkty przecięć, co sprawia, że finalna złożoność jest równa $O((n+k)\log n)$, gdzie k - ilość przecięć. Zaimplementowany przeze mnie algorytm korzysta z takiej samej struktury stanu jak poprzedni, jednak struktura zdarzeń jest lekko zmodyfikowana. Jako `[-1, [p.x, p.y], idx1, idx2]` oznaczam punkty przecięć odcinków o indeksach `idx1` i `idx2` w punkcie `[p.x, p.y]` które należą do struktury zdarzeń. Kroki wykonywane $2n+k$ razy są następujące:

1. Analogicznie jak 1. z poprzedniego algorytmu

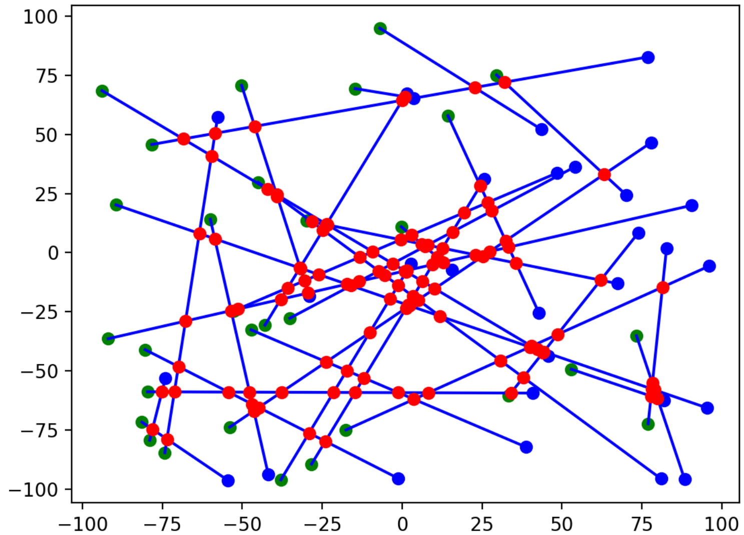
2. Analogicznie jak 2. z poprzedniego algorytmu

3. Jeżeli napotkamy punkt przecięcia odcinków o indeksach `idx1` i `idx2` to szukamy ich w miotle w $O(\log n)$ oraz zamieniamy te odcinki miejscami. Następnie sprawdzamy przecięcia zamienionych elementów z ich nowymi sąsiadami co dzieje się w czasie stałym $O(\text{const})$.

Stąd złożoność każdego kroku jest zachowana $O(\log n)$ oraz złożoność całego algorytmu wynosi $O((n+k)\log n)$. Algorytm zapewnia jednokrotne dodawanie przecięcia do zbioru przecięć, dzięki zbiorowi zawierającemu wszystkie przecięcia i dodawaniu tylko tych, których jeszcze nie ma. Główny algorytm zwraca zbiór przecięć jako jedynie pary indeksów segmentów, które się przecinają, jednak współrzędne punktu ich przecięcia wyliczyć w czasie liniowym do ilości przecięć. Poniżej w *Wizualizacji 5* i *Wizualizacji 6* zamieszczam dwa zbiory wraz z zaznaczonymi punktami przecięć oraz ilością znalezionych przecięć.

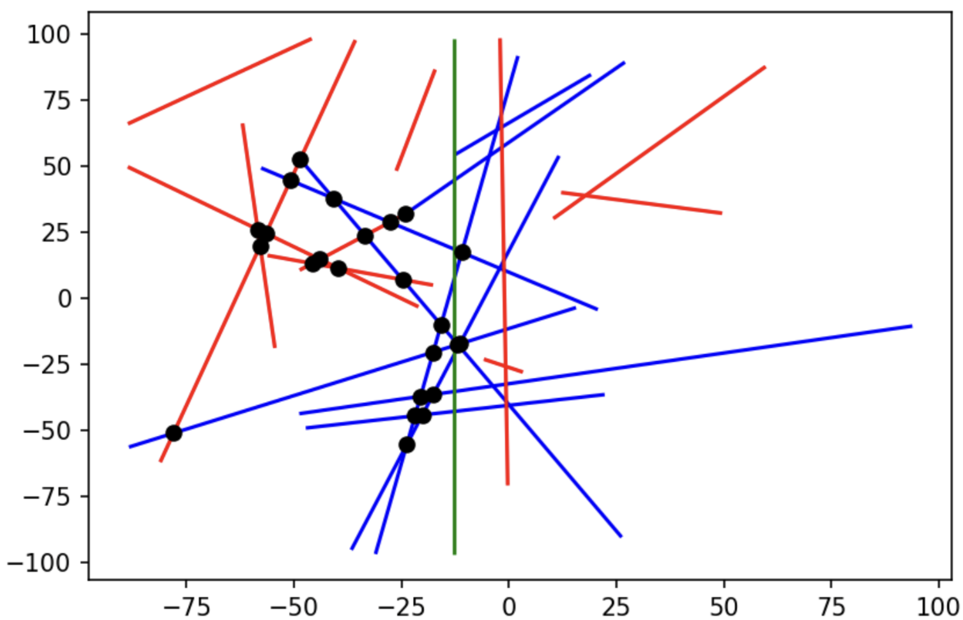


Wizualizacja 5: Znaleziono 9 przecięć.



Wizualizacja 6: Znaleziono 109 przecięć.

Poprawność sprawdziłem podejściem brute force. W *Wizualizacji 7* zamieściłem przykładowo klatkę z algorytmu znajdującego wszystkie przecięcia. Oznaczenia są analogiczne do *Wizualizacji 4*.



Wizualizacja 7: Graficzna reprezentacja znajdywanych przecięć.

Pomiar wydajności

W tabeli zamieściłem średnie czasy znalezienia wszystkich punktów przecięcia dla zbiorów o wielkości n:

n	czas
10	0.00015s
50	0.004s
100	0.03s
150	0.12s
200	0.4s
250	1.1s

Tabela 1: Pomiar czasu dla różnych zbiorów odcinków.