Kraków, 15.12.2020 r.

Paweł Kruczkiewicz,

Algorytmy geometryczne,

Grupa nr 8 (czwartek 16:15, tydzień B)

Sprawozdanie nr 4

Temat: Przecinanie się odcinków

Cel ćwiczenia

Zapoznanie się z algorytmem wyznaczania przecięć odcinków na płaszczyźnie omówionym na wykładzie, jak również w literaturze do przedmiotu.

Wstęp teoretyczny

Znajdowanie się przecinających odcinków jest ważnym w geometrii obliczeniowym zagadnieniem. Wykorzystuje się go m. in. w planowaniu ruchu robota, w problemie nakładania się map czy operacjach boolowskich w CAD.

Nasz problem zawężamy do zagadnienia – dla zbioru linii w przestrzeni dwuwymiarowej mamy znaleźć wszystkie przecięcia się odcinków oraz wskazać, które odcinki się przecinają. Naiwny algorytm sprawdzający wszystkie pary odcinków, działałby w czasie O(n^2), gdzie n to liczba linii w danym zbiorze. Jest to jednocześnie dolne ograniczenie czasu działania dla zbioru, gdzie liczba przecinających odcinków wynosi O(n^2) – program musi podać je wszystkie.

Opisany w sprawozdaniu program działa na zasadzie algorytmu zamiatania. Czas działania opisywanego algorytmu wynosi O((P+n)logn), gdzie P jest liczbą przecięć. Dokładny opis znajduje się w dalszej części sprawozdania.

Specyfikacja

Doświadczenie przeprowadzono na 64-bitowym systemie Windows 10 na 4-rdzeniowym procesorze firmy Intel o taktowaniu zegara 2.30 GHz. Kod oraz obliczenia wykonano w Jupyter Notebooku. Wersja pythona to 3.8. Precyzja obliczeń jest ograniczona przez zapis 64-bitowej liczby zmiennoprzecinkowej (odpowiednik typu double w językach C).

Jako wyznacznika użyto samodzielnie zaimplementowanego wyznacznika 3x3 o dokładności $\varepsilon = 10^{-13}$. Wyznacznik ten osiągnął najlepsze wyniki w testach opisanych w poprzednim sprawozdaniu.

Plan ćwiczenia

1. Przygotowanie procedury, pozwalającej wprowadzać w sposób interaktywny kolejne odcinki (reprezentowane przez pary wierzchołków), a także generować losowo zadaną liczbę odcinków z podanego zakresu współrzędnych 2D. Odcinki pionowe powinny być eliminowane i żadna para

odcinków nie powinna mieć końców odcinków o tej samej współrzędnej x. Program powinien umożliwiać zapis i odczyt zbioru odcinków.

2. Zaimplementowano algorytm zamiatania sprawdzający, czy choć jedna para odcinków w zadanym zbiorze się przecina.

3. Opisano w sprawozdaniu, jak została zaimplementowana struktura stanu (stan miotły) oraz struktura zdarzeń w Twoim programie.

4. Uzupełniono procedurę wykrywającą przecięcie o wizualizację kolejnych kroków (pozycja i stan miotły).

5. Przetestowano program na różnych zestawach danych.

6. Odpowiednio zmodyfikowano program - zaimplementowano algorytm wyznaczający wszystkie przecięcia odcinków. Na wyjściu program podaje liczbę wykrytych przecięć, współrzędne przecięć oraz dla każdego przecięcia odcinki, które się przecinają. Zwizualizowano program (bez potrzeby modyfikacji poprzedniego wizualizatora).

7. W sprawozdaniu napisano, czy konieczne były zmiany w strukturze danych oraz ich rodzaju. Odpowiedziano na pytanie, czy w przypadku obu algorytmów konieczne są takie same struktury zdarzeń. Odpowiedź uzasadniono.

8. W sprawozdaniu krótko opisano, jak obsługiwane są zdarzenia początku odcinka, końca odcinka i przecięcia odcinków z uwzględnianiem wybranych struktur danych.

9. Przetestowano zmodyfikowany program na różnych zestawach danych.

10. Znaleziono i wprowadzono taki układ odcinków, przy którym pewne przecięcia będą wykrywane więcej niż jeden raz, Sprawdzono czy, i jeśli tak, to jak program to uwzględnia. Napisano to w sprawozdaniu.

Ad. 1

Odcinki wprowadzane są za pomocą interfejsu graficznego. Losowanie odcinków zajmuje się funkcja „randomLines", przyjmująca 4 parametry współrzędnych oraz liczbę linii do wylosowania.

Funkcja usuwania niepasujących linii jest nieco ciekawsza. Punkty pionowe są usuwane w oczywisty sposób, natomiast linie o końcach o tych samych współrzędnych x są usuwane liniowologarytmicznie w następujący sposób: tworzymy tablicę odcinków uporządkowanych wg pierwszej współrzędnej x (tj. wspł. x pierwszego końca) i analogiczną drugą tablicę posortowaną wg drugiego x. Usuwamy z nich punkty o tych samych x. Na koniec przechodzimy po obu tablicach liniowo i przechodzimy po nich analogicznie jak w procedurze *merge* w mergesorcie, usuwając za każdym razem punkty z tą samą x-wą współrzędną. Rzutujemy obie tablice na dwa osobne zbiory i bierzemy ich część wspólną.

Linie zapisywane i wczytywane mogą być w formacie *.json*.

Ad. 2

Zaimplementowany algorytm sprawdzania, czy chociaż jedna para odcinków się przecina oparty o procedurę zamiatania polega na zapisywaniu w strukturze priorytetowej kolejki Q (jest to tzw. linia zamiatania) wszystkich linii w kolejności rosnących współrzędnych x końców odcinka (każdy odcinek

trafia do kolejki 2 razy). Obiekty w kolejce nazywamy zdarzeniami. Dodatkowo inicjalizujemy strukturę T, przechowującą wszystkie obecnie przecinane przez linię zamiatania linie, posortowaną względem obecnych współrzędnych y linii. Przechodzimy po kolejnych punktach z kolejki priorytetowej. Dla każdego punktu startowego z kolejki dodajemy punkt do struktury T oraz sprawdzamy, czy linia przecina się z którymkolwiek sąsiednim punktem ze struktury T. Jeżeli tak, zwracamy prawdę. Podobnie dla każdego punktu końcowego sprawdzamy, czy punkty sąsiednie punktu przecinają się nawzajem, a następnie usuwamy punkt. Jeżeli się przecinają, zwracamy prawdę. Opróżniwszy kolejkę, zwracamy fałsz.

To, czy 2 odcinki się przecinają sprawdzane jest w czasie O(1) za pomocą wyznaczników.

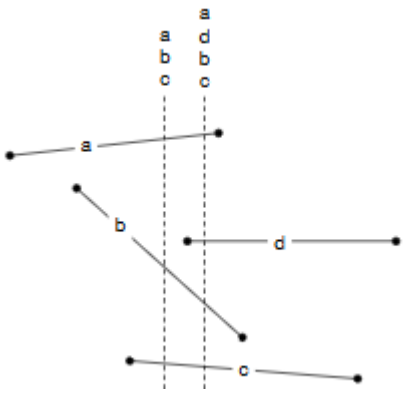
W implementacji zdecydowano się na umieszczenie wszystkich linii w obiektach klasy Line, zawierających oprócz punktów początku i końca odcinka, parametry linii , na której dany obiekt się znajduje, czyli A i B z równania y = Ax + B, a także obecną wartość współrzędnej y.

Zdarzenia zapisywane są w klasie Event, która w tej wersji algorytmu ogranicza się do zachowania punktu pozycji, w której się znajduje się owe zdarzenie, oraz linii, z której pochodzi.

Ad. 3

Używane w procedurze struktury danych to Q – priorytetowa kolejka zawierająca posortowane wg x zdarzenia. W implementacji skorzystano z PriorityQueue() z modułu queue standardowej biblioteki Pythona. Pozwoliło to na implementację wszystkich operacji kolejki (wstawianie w O(logn) oraz wyjmowanie w O(1)) dokładnie tak jak w opisanym wzorcowym algorytmie.

Druga struktura danych nastręczyła jednak o wiele więcej problemów. Algorytm wymaga, aby był to posortowany zbiór (np. zrównoważone drzewo BST), jednak posiadający wartości klucza, które są zmienne względem x, czyli y. Nie możemy przechowywać jedynie współrzędnych początkowych y danego odcinka, gdyż psułoby to algorytm w sytuacji przedstawionej na rys. 1 (odcinek d byłby niewłaściwie wstawiony). Możliwe jest stworzenie własnych implementacji zrównoważonego drzewa tudzież innego posortowanego słownika, który obliczałby obecne y w trakcie wstawiania nowego odcinka, co pozwoliłoby na wstawianie w czasie O(logn), jak wymaga tego algorytm. Pisanie takiego komparatora jest jednakże pracochłonne i niekonieczne do zrozumienia funkcjonowania algorytmu.



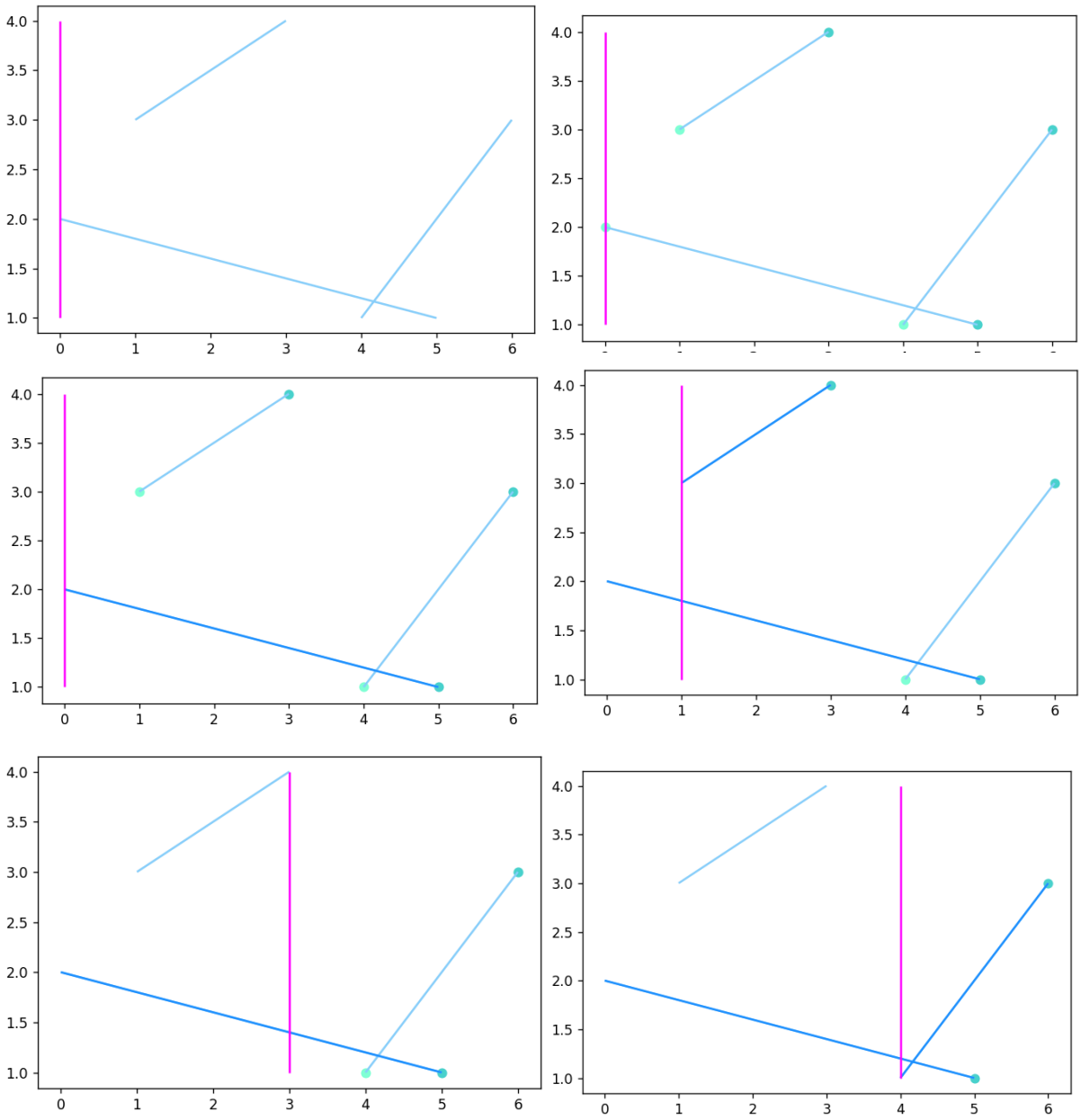
Rysunek 1 - przykład właściwego dodania punktu do struktury T

W implementacji zdecydowano się na użycie klasy SortedSet z modułu sortedcontainers, którego implementacja opiera się na drzewie czerwono-czarnym.

Struktura T sprawiła również inny problem, co zostało opisane w punkcie 7.

Ad. 4

Wizualizacja dla przykładowego zbioru danych (rys. 2):



Rys. 2 – przykładowe działanie programu. W pierwszej kolumnie inicjalizujemy kolejkę, dodajemy do niej wierzchołki. W drugiej kolumnie dodano pierwsze dwie linie. W trzeciej kolumnie zdjęto górny odcinek ze struktury T, a następnie, po dodaniu do drzewa ostatniej linii zauważono, że przecina się ona z jej „następnikiem”, więc przerwano dalsze sprawdzanie i zwrócono prawdę

Ad. 5

Testowane zbiory dostępne są w kodzie programu

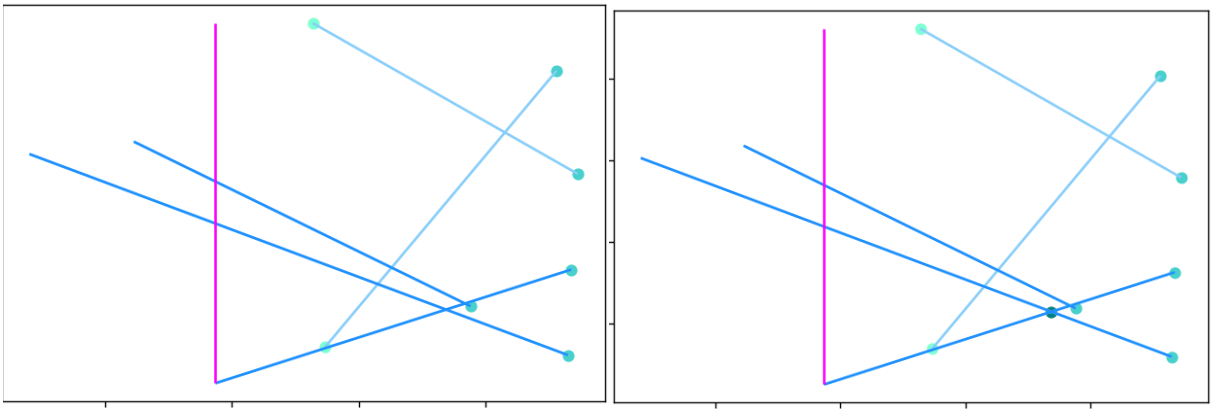
Ad. 6

Program znajdujący wszystkie przecięcia różni się od powyższego tym, że:

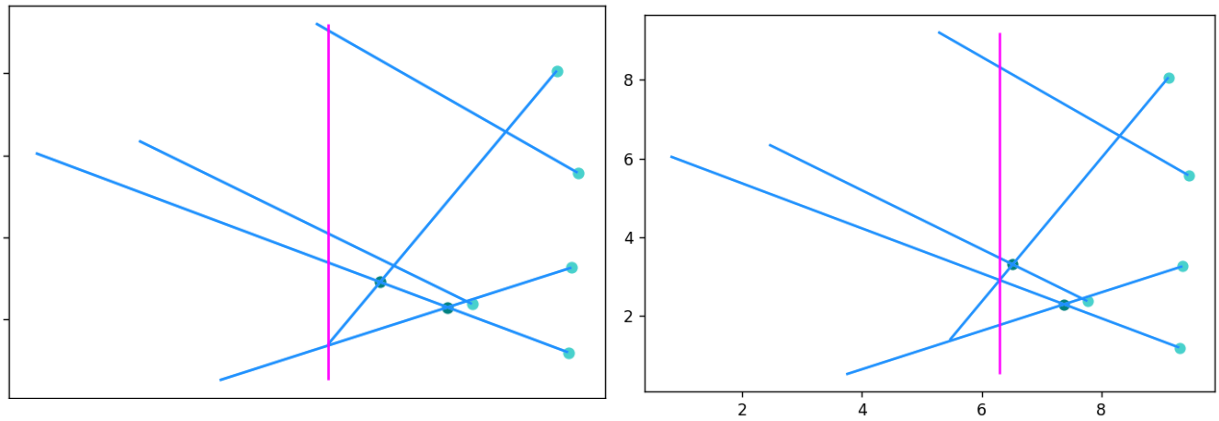
a) oprócz końców i początków odcinków zdarzeniami mogą być również przecięcia, które dodawane są do kolejki w momencie znalezienia

b) dla każdego zdarzenia będącego przecięciem, zamieniamy miejsce przeciętych linii w strukturze T, i sprawdzamy, czy nie przecinają się z nowymi sąsiadami.
c) Tuż przed przetworzeniem włożeniem przecięcia do kolejki sprawdzamy, czy dany punkt się nie powtórzył. To zostało zaimplementowane jako słownik, w którym kluczami są hasze indeksów linii przecięcia, a wartością – samo zdarzenie.

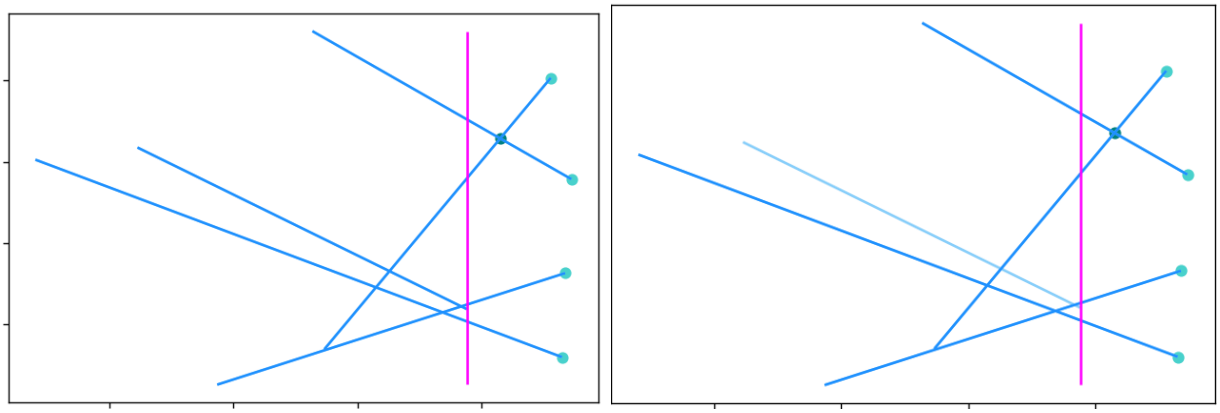
Wizualizacja (fragmenty):



Rys. 3 a – w momencie przecięcia, punkt przecięcia jest dodawany do kolejki



Rys. 3 b – w trakcie przejścia do nowego punktu przecięcia, kolejne punkty przecięcia są również dodawane (jeśli istnieją) Należy zauważyć, że w powyższym przypadku wcześniej dodany punkt znajdzie się w kolejce po raz drugi.



Rys. 3c – w tym kroku algprytm wykrywa punkt znajdujący się już za miotłą, jednak nie dodaje go ze względu na słownik haszujący zdarzenia po indeksach linii przecięcia

Program kończy działanie wraz z wyjęciem ostatniego punktu z kolejki.

Ad. 7

W porównaniu z algorytmem sprawdzającym czy istnieją punkty przecięcia, algorytm ich znajdowania potrzebował:

a) Nieco zmodyfikowanej klasy zdarzenia, która teraz zapamiętywała nie tylko położenie, ale i jego typ. Dla końców odcinków zapisuje linię, na której się znajduje, a dla punktów przecięcia – obie linie, do których należy przecięcie.
b) Dodano wspomniany wcześniej słownik do sprawdzania, czy dany pukt nie był już przetworzony.

Struktury stanu pozostały bez zmian. Należy jednak wspomnieć o drugiej trudności technicznej – zmienianiu kolejności odcinków w strukturze T. W momencie przecięcia ich klucze są jednakowe (y1 == y2). W celu naprawienia tego, wartości są wyliczane dla x + epsilon. Sprawia to niestety, ze dla niektórych zbiorów danych program zwracałby niewłaściwy wynik. Dodano zatem AssertionError dla linii dodanych nieprawidłowo. Program nie zwraca wtedy wyniku.

Oczywiście, w przypadku własnych struktur danych można by było ograniczenie to nieco zmienić i „dopasować" je do naszego problemu, np. poprzez rozwiązywanie „remisów" w trakcie wstawiania elementu.

Ad 8.

Zachowanie się zdarzeń zostało opisane wyżej.

Ad. 9

Testy znajdują się w kodzie omawianego programu.

Ad. 10

Podczas pisania programu założono, że ponad 3 linie w zbiorze nie przecinają się w tym samym punkcie. Sprawdziwszy, jak w takich sytuacjach zajmuje się napisana procedura, okazuje się, że punkty takie wykrywane są wielokrotnie.

Wnioski

Przedstawiony program pozwala na szybsze asymptotycznie sprawdzanie, ile, gdzie i jakie linie się przecinają. Nie jest on jednak prosty w implementacji i wymagania algorytmu wymuszają użycia nieco zmodyfikowanych podstawowych struktur danych, w które biblioteki standardowe nie zawsze są wyposażone.