Sprawozdanie z laboratorium 4. z algorytmów geometrycznych - Przecięcia odcinków

Hubert Miklas
Grupa 5, Poniedziałek 15:00-16:30
Data wykonania ćwiczenia:
25-11-2024
Data oddania sprawozdania:
12-12-2024

1 Wstęp do ćwiczenia

1.1 Specyfikacja środowiska

System: Windows 10 Home
Procesor: Intel(R) Core(TM) i5-7400 CPU 3.00 GHz
Pamięć RAM: 24 GB
Karta graficzna: NVIDIA GeForce GTX 1060
Środowisko: Jupyter Notebook, Python 3.12.8

1.2 Opis i cel ćwiczenia

Celem ćwiczenia jest wygenerowanie różnych zbiorów punktów na płaszczyźnie oraz wyznaczenie ich przecięć algorytmem szybszym niż podejście naiwne (przecinanie się każdego odcinka z każdym).

2 Wstęp teoretyczny

Problem przecinania odcinków polega na wyznaczeniu wszystkich punktów przecięcia dla zadanej kolekcji odcinków na płaszczyźnie.

W ramach zadania zaimplementowano dwa podejścia do tego problemu: algorytm naiwny oraz algorytm zamiatania.

2.1 Wykrywanie przecięć odcinków

W celu sprawdzenia, czy dwa odcinki ab i cd na płaszczyźnie się przecinają, zastosowano podejście geometryczne oparte na analizie orientacji punktów oraz współliniowości. Kluczowe kroki algorytmu to:

1. **Obliczenie orientacji punktów**: Wykorzystano wyznacznik z **Równania 1**, który pozwala określić względne położenie punktu c względem odcinka ab.

$$\textbf{det(a,b,c)} = \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix} \tag{1}$$

gdzie wynik dodatni oznacza, że c leży po jednej stronie ab, wynik ujemny po drugiej stronie, a zero oznacza współliniowość (w obliczeniach przyjęto tolerancję dla zera $\epsilon = 10^{-18}$).

2. **Test współliniowości**: Jeśli punkty c i d są współliniowe względem odcinka ab, sprawdzane jest, czy leżą one na odcinku ab. Wykorzystano w tym celu funkcję `in_segment`, która porównuje współrzędne punktów.

3. **Analiza położenia końców**: Sprawdzane jest, czy końce jednego odcinka leżą po przeciwnych stronach drugiego odcinka. Jeśli tak, odcinki się przecinają.

4. **Łączne warunki przecięcia**: Odcinki się przecinają, jeśli:

   - Punkty c i d leżą po przeciwnych stronach ab, oraz punkty a i b leżą po przeciwnych stronach cd,
   - Odcinki są współliniowe i mają wspólny punkt.

2.2 Podejście naiwne

Podejście naiwne opiera się na sprawdzaniu wszystkich par odcinków, co prowadzi do złożoności czasowej $O(n^2)$. Główne kroki algorytmu są następujące:

1. **Inicjalizacja**: Ustawienie zbioru wynikowego na pustą tablicę.

2. **Główna pętla**: Iteracja po wszystkich parach odcinków w zbiorze wejściowym.

   - Dla każdej pary odcinków sprawdzanie, czy się przecinają (użycie wyżej opisanej procedury weryfikacji przecięcia odcinków).
   - Jeżeli odcinki się przecinają, wyznaczany jest punkt przecięcia korzystając z równań kierunkowych prostych zawierających te odcinki.

3. **Zwrócenie wyniku**: Zwrócenie zbioru wszystkich punktów przecięcia odcinków wraz z ich numerami.

2.3 Algorytm zamiatania

Algorytm zamiatania wykorzystuje koncepcję "miotły" przemieszczającej się wzdłuż osi x w celu efektywnego wykrywania przecięć. Ma on złożoność czasową $O((n + P) \log n)$, gdzie P to liczba punktów przecięcia. Sąsiadami odcinka L definiujemy takie odcinki, dla których jeden ma najmniejszą wartość y większą od wartości y dla L w miejscu zdarzenia x i drugi, którego wartość y mniejsza od wartości y dla L w miejscu zdarzenia x jest największa.

2.3.1 Funkcje wykorzystane w algorytmie

1. Do sprawdzenia, czy odcinki nie były już rozważane służy Set (Python-owa mapa haszująca). Do tej struktury dodawana jest para indeksów oznaczających numery pozycji odcinków w tablicy zawierającej odcinki. Każda para to `(indeks_mniejszy, indeks_większy)`.

2. "Przesuwanie" miotły odbywa się przez zmianę statycznego `x` przypisanego do klasy `LineSegment` przy użyciu funkcji `set_x`, która jest zaimplementowana w każdej zaimplementowanej strukturze stanu.

2.3.2 Główna część algorytmu

1. **Inicjalizacja**: Stworzenie struktury zdarzeń (kolejki priorytetowej), zawierającej wszystkie punkty początku i końców odcinków oraz potencjalne punkty przecięcia. Inicjalizacja struktury stanu (`Drzewo Czerwono-Czarne` lub zmodyfikowana struktura `SortedSet`), przechowującej aktywne odcinki.

2. **Główna pętla algorytmu**: Pobieranie zdarzeń z kolejki w kolejności rosnących współrzędnych x i ich obsługa:

   - **Zdarzenie początku odcinka:** Miotła, czyli struktura zdarzeń jest przesuwana do aktualnego miejsca. Następnie do struktury stanu jest dodawany aktualny odcinek. Sprawdzamy, czy po takim dodanie nie zostało wykryte przecięcie z jego sąsiadami.
   - **Zdarzenie końca odcinka:** Przesunięcie miotły do aktualnego miejsca zdarzenia. Sprawdzamy, czy po usunięciu odcinka, w którym punkt zdarzenia ma koniec, jego sąsiedzi się ze sobą nie przetną. Następnie usuwamy ten odcinek se struktury stanu.
   - **Zdarzenie przecięcia:** Miotła jest przesuwana do punktu zdarzenia, którego współrzędne. Identyfikowani są sąsiedzi odcinków tworzących punkt przecięcia. Są to odcinki położone najbliżej powyżej i poniżej każdego z dwóch odcinków tworzących punkt przecięcia. W szczególności unika się traktowania drugiego odcinka jako sąsiada dla pierwszego i odwrotnie. Następnie oba odcinki, które tworzą punkt przecięcia, są usuwane ze struktury stanu. Po usunięciu, oba odcinki są ponownie wstawiane do struktury w zmodyfikowanej postaci, co odpowiada przesunięciu miotły do nowej pozycji (x) i uwzględnieniu zmieniających się relacji między odcinkami. W kolejnym kroku algorytm analizuje sąsiedztwo nowo wstawionych odcinków:
     - Dla pierwszego odcinka (`new_first_segment`) sprawdzane jest potencjalne przecięcie z jego dolnym sąsiadem (`second_segment_lower_neighbour`).
     - Dla drugiego odcinka (`new_second_segment`) sprawdzane jest potencjalne przecięcie z jego górnym sąsiadem (`first_segment_upper_neighbour`).

     Te nowe zdarzenia przecięć są dodawane do struktury zdarzeń przy jednoczesnym sprawdzaniu, czy takie przecięcie nie było już wcześniej uwzględnione (mechanizm oznaczania zdarzeń jako sprawdzone).

     Jeśli jest to algorytm sprawdzający, czy istnieją odcinki przecinające się, wystąpienie takiego zdarzenia (przecięcie) powoduje zwrócenie wartości `True`. W przeciwnym

wypadku zdarzenie jest dodawane do listy obsłużonych zdarzeń, która unika dodania tego samego punktu więcej niż jeden raz.

3. **Zwrócenie wyniku**:

   - W podstawowym algorytmie zwracane są punkty przecięcia z indeksami ich odcinków.
   - Jeżeli jest to algorytm sprawdzający, czy istnieje przecięcie, zwraca `False`, ponieważ nie zwrócił wcześniej `True`, czyli żadne odcinki się nie przecięły.

2.4 Komentarz porównania między algorytmem znajdywania przecięć i sprawdzenia, czy istnieją przecinające się odcinki

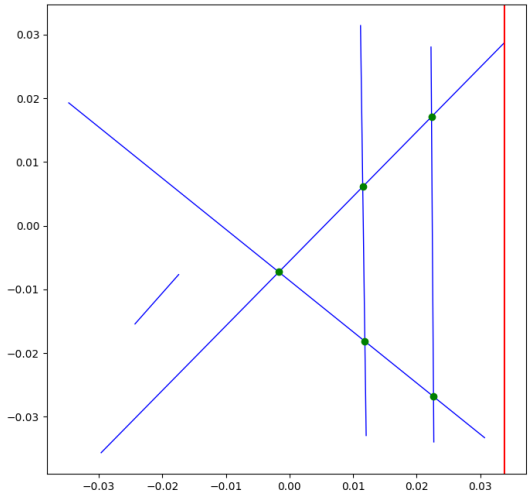
Do obydwu algorytmów wykorzystałem te same struktury danych. Do miotły, czyli struktury zdarzeń, wykorzystałem kopiec minimalny, ponieważ interesuje nas tylko punkt o najmniejszej współrzędnej x. Nie było potrzeby, żeby stosować bardziej zaawansowane struktury jak drzewa, ważne było, aby nowe punkty zdarzeń (punkty przecięcia) były efektywnie wstawiane na strukturę i zapytanie o nie było szybkie. Po rozważeniu każdego zdarzenia należy rozważyć następujące po nim zdarzenie, o najmniejszym x większym od rozpatrzonego zdarzenia.

3 Przebieg obliczeń

Algorytm został przetestowany pod kątem wykrywania przecięć w zadanych zbiorach testowych.

3.1 Test 1

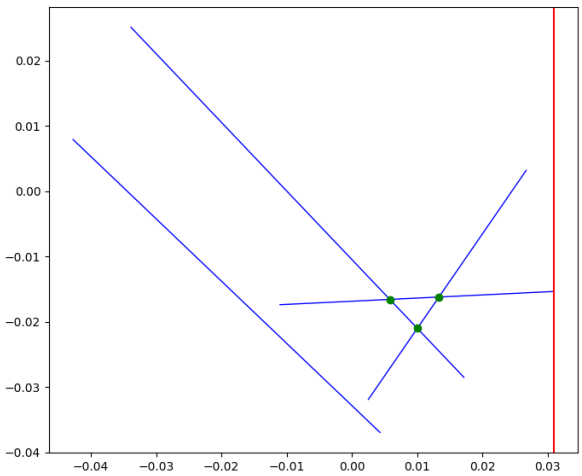
Ten test miał weryfikować, czy zamiana odcinków jest poprawnie obsługiwana. Po pierwszym punkcie przecięcia kolejność na strukturze stanu najdłuższych (widocznych na **Rysunku 1**.



Rysunek 1: Po przejściu przez pierwszy punkt przecięcia struktura stanu zamienia kolejność odcinków

3.2 Test 2

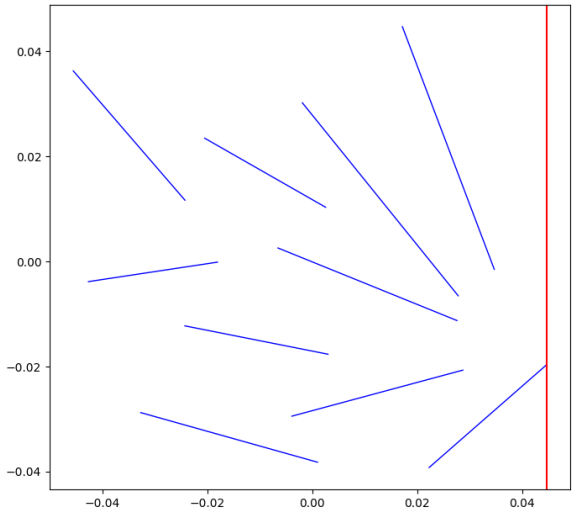
Test ten sprawdzał, czy po zamianie kolejności w strukturze stanu zostanie znaleziony ostatni punkt przecięcia. Na **Rysunku 2** widać, że algorytm zadziałał poprawnie.



Rysunek 2: Kolejny test

3.3 Test 3

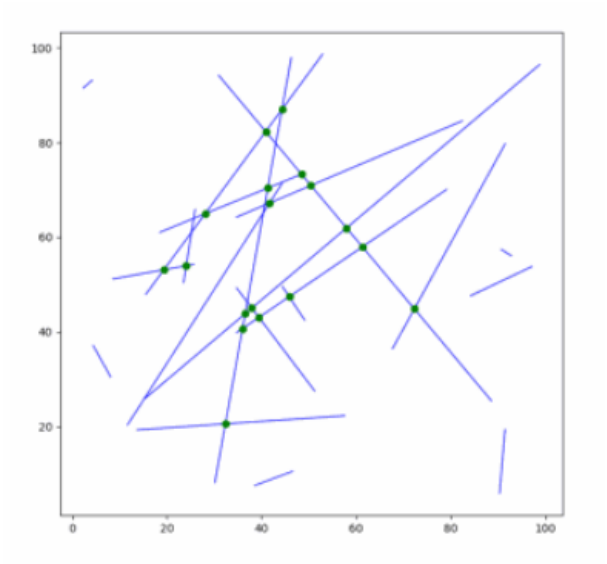
Ten test sprawdzał, czy algorytm nie wykrywa odcinków, które się nie przecinają. Na **Rysunku 3** widać, że algorytm zadziałał poprawnie.



Rysunek 3: Żadne odcinki się nie przecinają

3.4 Test 4

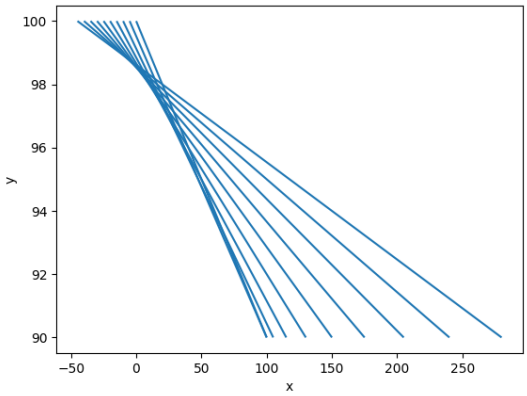
Ostatni test był losowym zbiorem odcinków. Na **Rysunku 4** widać, że algorytm zadziałał poprawnie.



Rysunek 4: Żadne odcinki się nie przecinają

4 Wnioski

Po przetestowaniu algorytmów widać, że oba działają poprawnie. Odpowiednie struktury stanu i zdarzeń pozwalają na osiągnięcie docelowej złożoności $O((n+P)\log n)$ gdzie n to liczba odcinków, a P to liczba ich przecięć. W pesymistycznym przypadku, gdzie przecięć jest rzędu $P = O(n^2)$, dla dużych n może okazać się, że algorytm naiwny jest minimalnie szybszy. Taki przypadek przedstawiono na **Rysunku 5**.



Rysunek 5: Pesymistyczny przypadek, w którym ilość przecięć odcinków $P = 1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{(n-1)\cdot n}{2} = O(n^2)$

Pełna wizualizacja testów znajduje się w plikach .gif.