Sprawozdanie z Implementacji Algorytmu Zamiatania Wykrywającego Przecięcia Odcinków

Artur Radwański - grupa 4

3 grudnia 2025

1 Realizacja Ćwiczenia

1.1 Cel ćwiczenia

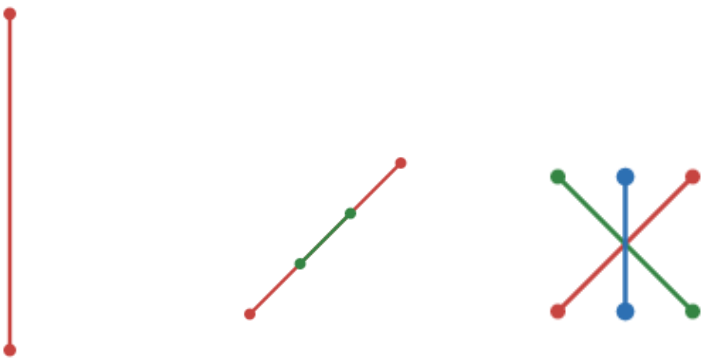
Celem ćwiczenia było zapoznanie się z algorytmem zamiatania wyznaczającym przecięcia się odcinków na płaszczyźnie kartezjańskiej, wraz z jego implementacją w dwóch wersjach. Jedna sprawdzała czy istnieje przecięcie w zbiorze odcinków, a druga znajdowała wszystkie przecięcia w danym zbiorze

1.2 Wstęp teoretyczny

Algorytmy zamiatania wykorzystują miotłę, strukturę stanu oraz strukturę zdarzeń. W przypadku tego algorytmu miotła, to pionowa prosta, struktura stanu, to lista odcinków przecinająca miotłę uporządkowana od najmniejszego y punktu przecięcia, struktura zdarzeń to lista punktów które są jednym z końców odcinka lub są miejscem przecięcia dwóch odcinków, uporządkowana od najmniejszego x każdego punktu.

O zbiorze odcinków dla którego wykonujemy algorytm zakładamy, że:

- żaden odcinek nie jest pionowy
- żadne dwa odcinki nie przecinają się w więcej niż jednym punkcie
- w jednym punkcie przecinają się co najwyżej dwa odcinki



(a) Odcinek pionowy (b) Odcinki się pokrywają (c) Przecięcie 3 odcinków

Rysunek 1: Wykluczone sytuacje

Na początku dodajemy do struktury zdarzeń wszystkie punkty które są początkami lub końcami odcinków. Po uruchomieniu algorytmu miotła porusza się w kierunku zamiatania (w tym przypadku, wzdłuż wektora $[1, 0]$), za każdym razem kiedy w strukturze stanu zmienia się kolejność odcinku, sprawdzamy czy nowi sąsiedzi (odcinki które wcześniej ze sobą nie sąsiadowały) się przecinają, jeśli tak zapisujemy punkt przecięcia i dodajemy go do struktury zdarzeń, w przeciwnym razie przechodzimy do następnego zdarzenia.

1.3 Opis implementacji

Struktura stanu oraz struktura zdarzeń zostały w tym programie zaimplementowane z pomocą struktury SortedSet z biblioteki sorted containers. Struktura ta tworzy rosnący ciąg i w momencie wstawiania nowego elementu zapobiega dodawaniu duplikatów oraz nie burzy i znajduje indeks dla elementu który nie zniszczy porządku. Dodawanie i usuwanie elementów można wykonać w czasie $O(\log n)$ natomiast sprawdzania czy element należy do zbioru dokonuje się średnio w czasie $O(1)$. Dzięki takiej implementacji cały algorytm ma złożoność $O((P+n)\log n)$ gdzie P - liczba przecięć, n-liczba odcinków. Położenie miotły jest zaimplementowane jako zmienna statyczna klasy reprezentującej odcinek.

Obsługa zdarzeń:

- **Początek odcinka** - dodajemy odcinek do struktury stanu i sprawdzamy czy przecina się z sąsiadami
- **Koniec odcinka** - sprawdzamy czy sąsiednie odcinki się ze sobą przecinają po czym usuwamy go ze struktury stanu
- **Przecięcie odcinków** - sprawdzamy czy nowi sąsiedzi się ze sobą przecinają, zdejmujemy dwa odcinki które teraz się przecinają ze struktury stanu, przesuwamy miotłę o epsilon w kierunku zamiatania, i wkładamy oba odcinki z powrotem do struktury stanu

Wykrywanie wszystkich przecięć nie wymagało wielkich zmian w strukturze zdarzeń przy zastosowanym podejściu. Jedyna różnica polega na tym, że w wersji algorytmu która znajduje wszystkie przecięcia, zdarzenie może zawierać indeks jednego lub dwóch odcinków, w zależności od rodzaju zdarzenia. W wersji która sprawdza czy przecięcie istnieje, nie rozważamy obsługi zdarzenia przecięcia dwóch odcinków, więc w każdym zdarzeniu znajduje się tylko jeden indeks.

Wykrywanie tego samego przecięcia spowolniłoby algorytm i zdegradowało wynik, aby się przed tym chronić, krotki indeksów przecinających się odcinków przechowujemy w zbiorze i sorawdzamy czy nowi sąsiedzi w strukturze zdarzeń się przecinają tylko jeśli już tego nie sprawdzaliśmy.

2 Wyniki