|  |  |
| --- | --- |
| Wojciech Michaluk | grupa nr 6 godz. 13:00-14:30 czw. A |

**Sprawozdanie z laboratorium nr 4: Przecinanie odcinków**

|  |  |
| --- | --- |
| Data wykonania: 30.11.2023r. | Data oddania: 15.12.2023r. |

**1. Cel ćwiczenia**

Celem ćwiczenia jest zapoznanie się z algorytmem zamiatania wyznaczającym przecięcia się odcinków na płaszczyźnie oraz jego implementacja w dwóch wariantach: sprawdzanie, czy jakakolwiek para odcinków się przecina oraz obliczanie wszystkich punktów przecięć odcinków,   
a także wizualizacja wyników.

**2. Wstęp teoretyczny**

Algorytm zamiatania w ogólności wykorzystuje tzw. miotłę oraz pomocnicze struktury: strukturę zdarzeń oraz strukturę stanu . W przypadku problemu rozważanego w ramach tego laboratorium, miotłą jest prosta równoległa do osi .

**Założenia, które czynimy w ramach rozwiązania problemu:**

* żaden odcinek nie jest pionowy (równoległy do osi ) ani nie ma tej samej współrzędnej   
  x-owej końca odcinka co koniec innego odcinka
* dwa odcinki przecinają się w co najwyżej jednym punkcie (nie jest możliwa sytuacja, że pokrywają się – czy to całkowicie, czy częściowo)
* w jednym punkcie przecięcia przecinają się nie więcej niż dwa odcinki

Obraz zawierający linia, diagram

Opis wygenerowany automatycznie

**Rys. 2.1** Powyższe sytuacje są „zabronione”

Kierunek zamiatania jest wzdłuż osi (w stronę rosnących x-ów). Przesuwanie miotły następuje   
w tym kierunku. Jako stan miotły rozumiemy zbiór odcinków przecinających miotłę. Miotła zatrzymuje się w punktach zdarzeń, którymi są końce odcinków oraz wykryte punkty przecięć.   
To właśnie tutaj następuje aktualizacja stanu miotły oraz testy przecięć.

Informacje o zdarzeniach przechowujemy w strukturze zdarzeń, z kolei informacje potrzebne do obliczeń przechowujemy w strukturze stanu. Struktura stanu jest aktualizowana w każdym zdarzeniu. Na „zamiecionym” obszarze (czyli na lewo od miotły) znane jest rozwiązanie badanego problemu dla zdarzeń należących do tego obszaru, natomiast przecięcia na prawo od miotły są nieznane.

|  |  |
| --- | --- |
|  | **Rys. 2.2** Zdarzenia dla podanego przykładowego zbioru odcinków. Na czerwono zaznaczone są punkty przecięć, na niebiesko – końce odcinków. |

W każdym położeniu miotły wyróżniamy odcinki:

* przetworzone – oba ich końce znajdują się na lewo od miotły
* aktywne – aktualnie przecinają miotłę
* oczekujące – o obu końcach na prawo od miotły

|  |  |
| --- | --- |
| **Rys. 2.3** Odcinki oraz są *przetworzone*. Odcinki *aktywne*  w tym przypadku to , i . Odcinkami *oczekującymi* są oraz . |  |

**3. Metodologia, specyfikacja narzędzi i sprzętu**

Aby móc wprowadzać odcinki w sposób interaktywny przy użyciu myszki, skorzystałem z pakietu funkcji oferowanych przez bibliotekę matplotlib. Wykresy przedstawiające przykładowe   
i „wygenerowane” zbiory odcinków oraz wizualizacja poszczególnych kroków algorytmu zamiatania powstały przy użyciu biblioteki matplotlib oraz dzięki narzędziu przygotowanemu przez koło naukowe Bit.

Struktury pomocnicze do algorytmu zamiatania (zdarzeń, i stanu, ) zostały zaimplementowane  
z wykorzystaniem struktury **SortedSet** z zewnętrznego pakietu *sortedcontainers*. Jest to posortowany zbiór, który zapewnia obsługę podstawowych operacji potrzebnych do algorytmu zamiatania (wstawianie elementu, usuwanie ze struktury) w czasie , gdzie n oznacza liczbę odcinków w wejściowym zbiorze – jest to pożądana złożoność.

Aby móc poprawnie obsługiwać dodawanie odcinków do struktury stanu i usuwanie ich z niej, stworzyłem dodatkowe klasy **Point** oraz **Section**. Wynika to ze specyfiki SortedSet – więcej szczegółów w pliku z implementacją.

Program (w pliku „michaluk\_kod\_4.ipynb”) jest napisany w języku Python w środowisku Jupyter Notebook. Przedstawione wyniki pochodzą z uruchomienia programu na komputerze z systemem Windows 11 i procesorem Intel Core i5-8300H 2.30 GHz.

**4. Program ćwiczenia**

Na początku zaimplementowałem procedury pozwalające w sposób interaktywny wprowadzać odcinki poprzez zaznaczanie punktów myszką / generować zadaną liczbę odcinków o współrzędnych z podanego zakresu. Ich dokładny opis (m. in. „instrukcje” wprowadzania odcinków) znajduje się   
w pliku z implementacją.

Drugą częścią laboratorium jest implementacja algorytmu zamiatania w dwóch wariantach: aby sprawdzić, czy w zbiorze odcinków występuje jakiekolwiek przecięcie oraz aby znaleźć wszystkie przecięcia w zbiorze odcinków. W obu przypadkach użyłem wspomnianej struktury **SortedSet**.

W programie dla obu wersji zagadnienia wykorzystałem te same struktury zdarzeń i stanu. Istotna różnica polega na tym, że w pierwszym przypadku po wykryciu punktu przecięcia algorytm kończył działanie, zwracając stosowną wartość, a w drugim wariancie ten punkt był dodawany do struktury zdarzeń. Wydaje mi się, że nie ma konieczności używania tej samej struktury zdarzeń - można użyć innej struktury zdarzeń, ale niezależnie od wersji problemu w strukturze zdarzeń znajdują się wszystkie końce odcinków, które muszą być posortowane, więc rząd złożoności powinien być taki sam, jak przy użyciu SortedSet, czyli .

Obsługa zdarzeń:

* początku odcinka – odcinek jest dodawany do struktury stanu .
* końca odcinka – odcinek jest usuwany ze struktury stanu
* przecięcia odcinków – kolejność przecinających się odcinków w strukturze stanu jest zamieniana

Zdarzenie jest pobierane na początku każdej iteracji głównej pętli ze struktury , zdarzenia są ułożone zgodnie z przyjętą konwencją kierunku zamiatania (w kierunku rosnących x-ów).

Wstawianie do struktury stanu odbywa się z zachowaniem porządku – odcinki są ułożone rosnąco względem współrzędnej y-owej (z którą jest powiązany każdy aktywny odcinek).

We wszystkich przypadkach po wykonaniu operacji badamy pary nowych „sąsiadów”, szukając potencjalnych punktów przecięcia.

**5. Testowane zbiory danych i wyniki działania algorytmów**

Aby testy były rzetelne, starałem się dobrać zbiory odcinków różnej natury. Na następnych stronach przedstawiam wybrane **cztery** zbiory odcinków, dla których przetestuję działanie zaimplementowanych w ramach tego laboratorium algorytmów. Wśród nich jest zbiór, w którym jest duża liczba przecięć; zbiór z małą liczbą przecięć; zbiór, w którym nie ma przecięć oraz zbiór, w którym odcinki są dobrane w taki sposób, że przecięcia mogłyby być wykrywane wielokrotnie.

|  |  |
| --- | --- |
|  |  |
| **Rys. 5.1** Pierwszy testowany zbiór, z dużą liczbą przecięć – zbiór A | **Rys 5.2** Drugi testowany zbiór, z małą liczbą przecięć – zbiór B |
|  |  |
| **Rys. 5.3** Trzeci testowany zbiór, nie ma żadnego przecięcia – zbiór C | **Rys. 5.4** Czwarty testowany zbiór, sprawdzający potencjalne kilkukrotne znalezienie punktu przecięcia – zbiór D |

W poniższej tabeli przedstawiam wyniki działania funkcji sprawdzającej, czy istnieje co najmniej jedno przecięcie odcinków w zbiorze – True, jeśli w zbiorze występuje co najmniej jedno przecięcie, False w przeciwnym przypadku.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Testowany zbiór | Zbiór A | Zbiór B | Zbiór C | Zbiór D |
| Występuje przecięcie? | True | True | False | True |

**Tabela 5.5** Sprawdzenie występowanie przecięcia odcinków w testowanych zbiorach

Jak widać w tabeli 5.5, dla zbioru C słusznie program nie znalazł żadnego przecięcia – takowego nie ma. W pozostałych przypadkach znaleziono przecięcie. Co do zbioru D, czy w moim programie jedno przecięcie będzie wykryte wielokrotnie?

Przeanalizujmy tę sytuację. Dla układu odcinków podobnego jak poniżej (który kilkukrotnie się powtarza w zbiorze D) jedno przecięcie może być wykrywane (a co niepożądane – dodawane do struktury zdarzeń) wielokrotnie.

Obraz zawierający linia

Opis wygenerowany automatycznie

**Rys. 5.6** Problematyczny układ odcinków – ale czy dla mojego algorytmu?

Punkt przecięcia odcinków i będzie wykryty trzykrotnie - przy dodawaniu lewego końca ; gdy usuwamy prawy koniec oraz gdy usuwamy prawy koniec . **Czy mój program to uwzględnia?**

Tak. Korzystam ze zbioru (), w którym zapisuję już sprawdzane pary odcinków, odnoszących się do ich indeksów. Tak jak ułożenie odcinków w trakcie wykonania programu się zmienia, tak ich indeksy z oryginalnej listy odcinków są niezmienne, więc żadna para odcinków nie będzie sprawdzana dwukrotnie. Wadą takiego rozwiązania jest pesymistyczna złożoność pamięciowa (jeżeli zostanie zapisana (prawie) każda para odcinków) - .

**Algorytm zamiatania dla znajdowania wszystkich przecięć**

Dla każdego zbioru odcinków przedstawię teraz wybrane etapy algorytmu zamiatania oraz zamiecione zbiory. Konwencja kolorowania: punkty oraz odcinki mogą być zaznaczone kolorem

* niebieskim – są to odcinki oczekujące
* zielonym – są to odcinki aktywne
* żółtym – aktywne odcinki, które są w danej chwili sprawdzane
* brązowym – odcinki przetworzone

Ponadto miotła jest zaznaczona kolorem czerwonym.

Animacja przedstawiająca wszystkie kroki algorytmu zamiatania, wraz z efektem końcowym, są przedstawione w pliku z implementacją. W sprawozdaniu umieszczam również informacje o liczbie znalezionych przecięć w każdym ze zbiorów.

**Zbiór A**

|  |  |
| --- | --- |
|  |  |
|  |  |

**Rys. 5.7** Wybrane etapy algorytmu zamiatania dla zbioru A

Obraz zawierający linia, diagram, Wykres, zrzut ekranu

Opis wygenerowany automatycznie

**Rys. 5.8** Zamieciony zbiór A (17 przecięć)

**Zbiór B**

|  |  |
| --- | --- |
|  |  |
|  |  |

**Rys. 5.9** Wybrane etapy algorytmu zamiatania dla zbioru B

Obraz zawierający linia, diagram, zrzut ekranu, Wykres

Opis wygenerowany automatycznie

**Rys. 5.10** Zamieciony zbiór B (3 przecięcia)

**Zbiór C**

|  |  |
| --- | --- |
|  |  |
|  |  |

**Rys. 5.11** Wybrane etapy algorytmu zamiatania dla zbioru C

Obraz zawierający linia, diagram, Wykres, zrzut ekranu

Opis wygenerowany automatycznie

**Rys. 5.12** Zamieciony zbiór C (0 przecięć)

**Zbiór D**

|  |  |
| --- | --- |
|  |  |
|  |  |

**Rys. 5.13** Wybrane etapy algorytmu zamiatania dla zbioru D

Obraz zawierający linia, diagram, Wykres, Równolegle

Opis wygenerowany automatycznie

**Rys. 5.14** Zamieciony zbiór D (3 przecięcia)

**6. Podsumowanie i wnioski**

Testowane zbiory odcinków miały różną specyfikę, aby móc przetestować program w różnych warunkach. Zarówno dla przyjętych zbiorów testowych, jak i dla zbiorów w testach zaproponowanych przez Koło Naukowe AGH Bit, program radził sobie dobrze, co widać także na rysunkach i gifach wygenerowanych przez program. Program bazował na opisie algorytmu zamiatania przedstawionym na wykładzie.

Struktury zdarzeń i stanu można zaimplementować także w inny sposób. Modelowym rozwiązaniem jest użycie zrównoważonych (i/lub wzbogaconych) drzew BST, zapewnia to optymalną złożoność oraz przysparza mniej potencjalnych problemów przy „aktualizacji” współrzędnej y-owej odcinka, kiedy zmienia się współrzędna x-owa.

Złożoność algorytmu zamiatania to wtedy (dla mojej implementacji również) , gdzie oznacza liczbę odcinków, a – liczbę przecięć. Z tego powodu, pesymistycznie złożoność czasowa jest gorsza niż złożoność algorytmu „Bruteforce”, która jest oszacowana na , bowiem liczba przecięć może być kwadratowa względem liczby odcinków. Jednakże, zwykle liczba przecięć jest zdecydowanie mniejsza.