Algorytmy geometryczne

Sprawozdanie - ćwiczenie 3.

Dominik Adamczyk

Gr. 2

1. Specyfikacja techniczna urządzenia, na którym wykonano ćwiczenie

- System: Windows 11 Home x64
- Procesor: Intel Core i5-1135G7 2.4MHz – 4.2 MHz
- Pamięć ram: 16 Gb
- Język programowania: Python 3.11 z bibliotekami numpy, matplotlib, json, sortedcontainers, random
- Środowisko: Jupyter Notebook z użyciem narzędzia załączonego na potrzeby laboratoriów

2. Treść i cel ćwiczenia

Ćwiczenie polegało na zaimplementowaniu algorytmu zamiatania pozwalającego sprawdzić istnienie przecięć w zadanym zbiorze, a później zmodyfikowanie go tak aby wyznaczał wszystkie punkty przecięć między odcinkami.

3. Przystosowanie aplikacji

Do wykonania ćwiczenia pomocne było zaimplementowanie dodatkowych funkcji pozwalających wczytywać i zapisywać wykres do pliku, a także umożliwiających ekstrakcję linii narysowanych na wykresie, na których w późniejszym etapie (po odpowiednich konwersjach) wykonywane są algorytmy. Dodana została również funkcja tworząca czyste płótno (z możliwością zadania jego wymiarów), na którym można narysować odcinki rozpatrywane w późniejszych algorytmach.

4. Ograniczenia algorytmu, generowanie odcinków

Implementowane algorytmy nie funkcjonują poprawnie gdy zadane odcinki są pionowe, bądź istnieją dwa końce odcinków o takiej samej współrzędnej x. Z tego powodu odcinki generowane losowo są każdorazowo sprawdzane pod kątem tych ograniczeń i gdy nie spełniają wymagań są odrzucane. Zaimplementowana funkcja generująca odcinki pozwala określić ilość odcinków do wygenerowania, przedział na którym są generowane, a także maksymalną długość wygenerowanego odcinka – może być to przydatne w przypadku testów sprawdzających algorytm dla dużej liczby odcinków, które mają niewiele punktów przecięć.

5. Reprezentacja punktu, odcinka i zbioru odcinków

Do reprezentowania istotnych dla algorytmu elementów wykorzystuję podejście obiektowe. Jest to szczególnie przydatne, gdyż każdy z obiektów może przechowywać w sobie dodatkowe informacje (np. punkt, który ma informację o odcinku do którego należy) potrzebne do działania algorytmu.

Podstawowym obiektem jest punkt (Point()). Przechowuje on informacje:
- o wartości x i y
- type – typ punktu w zależności od którego podejmowane będą odpowiednie działania w algorytmie. Możliwe typy: początek odcinka, koniec odcinka, przecięcie odcinków.
- edgeIdx – indeks krawędzi (będącej w liście krawędzi reprezentującej ich zbiór) do której należy punkt (jeżeli jest to początek, albo koniec odcinka)
- crsIdxs – krotka z dwoma indeksami krawędzi. Punkt z tym polem jest przecięciem dwóch

odcinków, a indeksy z krotki na nie wskazują.
- getIter() – zwraca punkt w postaci interpretowalnej przez narzędzie graficzne.

Kolejnym zdefiniowanym obiektem jest odcinek Segment(). Składa się on z dwóch punktów, które są przekazywane w konstruktorze. Przekazywane punkty mają od razu wypełniane pola, tak aby określić, który z nich jest początkiem, a który końcem odcinka – początkiem jest punkt z mniejszą wartością x, a końcem punkt z większą. Poza tym odcinek zawiera pola i metody:
- start, end – punkt początkowy i końcowy wyznaczające odcinek.
- idx – informacja o indeksie krawędzi w liście.
- sortValue – wartość, po której sortowane będą odcinki przez które przechodzić będzie miotła. Wartość ta będzie zmieniana, gdy miotła będzie przechodziła po kolejnych zdarzeniach, a rozpatrywany odcinek będzie aktualnie przetwarzany przez miotłę.
- a, b– współczynnik kierunkowy i wyraz wolny prostej zawierającej w sobie odcinek.
- setIdx() – metoda ustawiająca pole idx odcinka i pola idx punktów ją tworzących.
- inBoundries() – metoda sprawdzająca, czy zadany punkt znajduje się wewnątrz prostokąta zadanego przez punkt początkowy i końcowy odcinka.
- getValY() – metoda zwracająca wartość funkcji liniowej (na podstawie a i b) w zadanym punkcie.
- intersection() – metoda, która po przekazaniu drugiego obiektu zwraca punkt przecięcia dwóch odcinków, a jeżeli to nie istnieje – wartość None. Punkt przecięcia odcinków wyznaczany jest w oparciu o równania prostych zawierających te odcinki (przy użyciu wartości a i b), oraz metodę inBoundries() użytą dla obydwu odcinków.
- getIter() – metoda zwraca odcinek w postaci, którą może zinterpretować narzędzie graficzne.

Obiektem odpowiadającym za główne działanie programu jest SegmentCloud(). Zawiera on zbiór odcinków, których przecięcia będą sprawdzane. Także w tym obiekcie, jako jego metody zaimplementowane są algorytmy z treści ćwiczenia. W konstruktorze obiekt ten przyjmuje listę odcinków (w formie tablicy, w której odcinek zadany jest jako para krotek współrzędnych informujących o punktach końcowych odcinka). Tak przekazany zbiór jest przetwarzany na opisane powyżej obiekty. Podczas przetwarzania odcinki klasy Segment() umieszczane są w tablicy, a odcinkom i ich punktom przypisywane są indeksy mówiące o ich pozycji w tej tablicy. Pola i metody obiektu klasy SegmentCloud():
- segments – tablica z odcinkami, których przecięcia będą badane.
- points – tablica z punktami będącymi początkami i końcami odcinków.
- out – wynik działania algorytmu znajdowania przecięć. Pole jest uzupełniane po przeprowadzeniu algorytmu i zawiera liczbę punktów przecięć oraz ich pozycje z parami odcinków tworzących dane przecięcie.
- minY, maxY – najmniejsza i największa wartość y wśród punktów z tablicy points (pola potrzebne do przeprowadzenia wizualizacji miotły).
- findIntersectionsBasic() – metoda znajdująca punkty przecięcia naiwnym, kwadratowym algorytmem – używana do sprawdzania poprawności algorytmu zamiatania.
- hasIntersections(), hasIntersectionsAnimation() – metody pierwszego z zadanych algorytmów zamiatania, sprawdzające czy istnieje przecięcie odcinków w zadanym zbiorze. Pierwsza metoda zwraca wartość boolowską, druga animację.
- intersections(), intersectionsAnimation() – metody implementujące główny algorytm zamiatania. Obydwie uzupełniają pole out, a pierwsza z nich zwraca zawartość tego pola i wykres z zaznaczonymi punktami przecięć. Druga zwraca kompletną animację działania algorytmu.
- savePointsToFile() – zapisuje zawartość pola out do wskazanego pliku.
- updateSweep() – metoda aktualizuje wartości punktów przecięcia odcinków przecinających się z

miotłą, gdy ta zmienia swoje położenie.
- getLC(), getPC(), getSweep() – metody zwracają obiekty, które dodawane są do wizualizacji.

6. Struktura stanu i zdarzeń dla implementowanych algorytmów.

Pierwszy z implementowanych algorytmów miał za zadanie zwracać informację, czy w zadanym zbiorze odcinków istnieje jakiekolwiek przecięcie. W tym celu konieczne było posłużenie się odpowiednimi strukturami danych. W moim wypadku strukturami zarówno stanu jak i miotły były obiekty klasy SortedSet() zaimportowane z biblioteki sortedcontainers. Struktura ta oparta jest na zrównoważonym drzewie binarnym i reprezentuje zbiór haszowalnych obiektów, którym można zadać pewien porządek. Dlatego, że jej implementacją jest drzewo binarne, to struktura pozwala na wykonywanie operacji wstawiania i usuwania elementów w czasie logarytmicznym, a przez haszowalność obiektów tej struktury pozwala na sprawdzanie ich wartości w czasie stałym. Dzięki tym własnościom posortowany zbiór dobrze sprawdza się w roli struktury miotły, do której w każdym kroku działania uproszczonego algorytmu będą wstawiane/usuwane kolejne odcinki, a ich kolejność będzie zachowana zgodnie z zadanym porządkiem – dla pierwszego algorytmu wystarczające jest porządkowanie odcinków po wartości y przecięcia odcinka z miotłą, dla drugiego algorytmu konieczne będzie dodatkowe porządkowanie po współczynniku kierunkowym (w przypadku, gdy rozważamy zdarzenie będące przecięciem odcinków).

W pierwszym algorytmie za strukturę zdarzeń również przyjąłem posortowany zbiór, chociaż tak złożona struktura nie jest konieczna. Jako, że pierwszy algorytm ma wykrywać jedynie istnienie przecięcia, to nie muszą w nim być rozpatrywane zdarzenia, które pojawiają się podczas działania algorytmu. Z tego powodu wystarczającą strukturą zdarzeń dla pierwszego algorytmu byłaby posortowana lista zawierająca wszystkie punkty początkowe i końcowe rozpatrywanych odcinków. Jako, że nie są dodawane nowe zdarzenia, to niepotrzebne są operacje wstawiania/usuwania, które umożliwia SortedSet. Użycie tej struktury zostało podyktowane tym, że od razu po zainicjowaniu zdarzeń te są posortowane w odpowiedniej kolejności.

W przypadku drugiego algorytmu do reprezentacji struktury zdarzeń i struktury stanu został użyty ten sam SortedSet(), jednakże tutaj jego użycie w przypadku struktury zdarzeń jest w pełni uzasadnione – podczas wykonywania algorytmu dodawane są nowe zdarzenia (będące punktami przecięcia odcinków), które muszą być wstawione zgodnie z ustalonym porządkiem (po rosnących wartościach x punktów).

7. Działanie algorytmu znajdowania wszystkich przecięć

Implementowany algorytm znajdowania wszystkich przecięć zaczyna się od zainicjalizowania struktury stanu i struktury zdarzeń, w której od razu umieszczane są wszystkie krańce odcinków. Następnie ze struktury zdarzeń usuwane są punkty o najmniejszej wartości x. Każdorazowo podczas wyciągnięcia punktu ze struktury zdarzeń aktualizowane są wartości przecięcia aktywnych odcinków z miotłą – chociaż nie zmienia to kolejności odcinków w zbiorze (poza przypadkiem zdarzenia będącego przecięciem odcinków), to jest konieczne do prawidłowego wstawiania nowych odcinków do zbioru. Każdy punkt należy do jednej z trzech kategorii i w zależności od tego, do której należy wykonywane będą różne kroki.

Jeżeli wyciągnięty punkt jest punktem początkowym, to odcinek, który zaczyna się od tego punktu jest wstawiany do struktury stanu. Po wstawieniu następuje sprawdzenie, czy wstawiony odcinek ma punkt przecięcia z odcinkiem położonym bezpośrednio nad nim i

odcinkiem położonym bezpośrednio pod nim (jeżeli te istnieją). Jeżeli jakikolwiek z tych punktów przecięcia istnieje, to jest on wstawiany do struktury zdarzeń.

Gdy wyciągany jest punkt końcowy danego odcinka, to algorytm sprawdza czy istnieje przecięcie odcinka leżącego bezpośrednio nad rozpatrywanym odcinkiem z odcinkiem występującym bezpośrednio pod nim (jeżeli takie istnieją). W przypadku zaistnienia takiego przecięcia jest ono dodawane do zbioru zdarzeń. Odcinek, którego punkt końcowy rozpatrywano zostaje usunięty ze struktury stanu.

Dla zdarzenia będącego punktem przecięcia dwóch odcinków przeprowadzana jest zamiana kolejności tych odcinków w strukturze stanu. Jako, że obydwa odcinki w punkcie przecięcia mają tą samą wartość y, to ustalenie ich nowej kolejności odbywa się przy pomocy ich współczynników kierunkowych (które stanowią jedno z pól obiektu klasy Segment()). Odcinek z większym współczynnikiem będzie znajdował się nad odcinkiem z współczynnikiem mniejszym. Następnie sprawdzane jest, czy istnieją punkty przecięcia pomiędzy odcinkiem z większym współczynnikiem kierunkowym, a odcinkiem bezpośrednio nad nim i pomiędzy odcinkiem z mniejszym współczynnikiem kierunkowym, a odcinkiem bezpośrednio pod nim. Istniejące punkty przecięcia umieszczane są w strukturze zdarzeń.

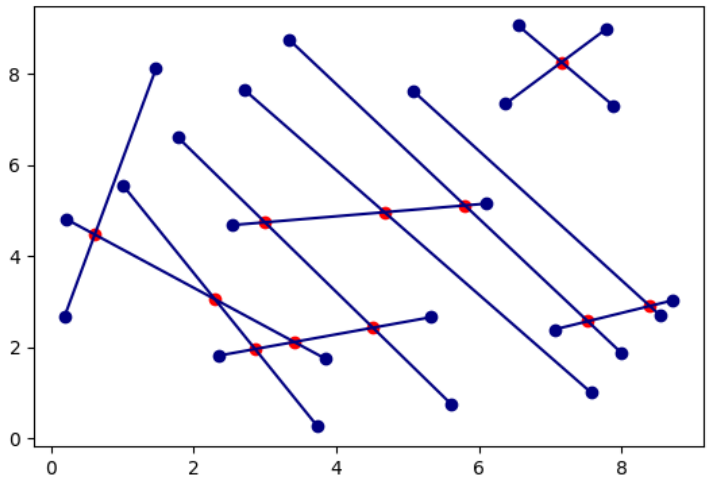
Za każdym razem gdy znajdowany jest punkt przecięcia między dwoma odcinkami algorytm sprawdza, czy ten punkt nie został znaleziony wcześniej. Aby to zrobić w algorytmie przechowywane są dwa zbiory punktów przecięć. Pierwszy z nich jest listą, do której dodawane są nowe punkty przecięcia (będące obiektami klasy Point()). Drugi natomiast jest zbiorem (obiektem klasy set()), w którym informacja o punktach przecięcia reprezentowana jest w postaci krotek z indeksami odcinków tworzących te przecięcia (indeksy są posortowane rosnąco). Dzięki temu możliwe jest sprawdzenie w czasie stałym, czy dane przecięcie zostało już uwzględnione przez algorytm, co pozwala zapobiec błędom wykonywania się programu. Jeżeli rozpatrywane przecięcie nie figuruje w zbiorze z indeksami odcinków, to zostaje ono dodane listy punktów, zbioru indeksów, a także struktury zdarzeń.

Algorytm implementowany jako pierwszy – odpowiadający na pytanie, czy jakiekolwiek odcinki się przecinają, to tak naprawdę zredukowana wersja przedstawionego powyżej algorytmu, która uwzględnia tylko przypadki zdarzeń będących początkiem i końcem odcinka. Jeżeli ten algorytm natrafia na przecięcie, to kończy działanie.

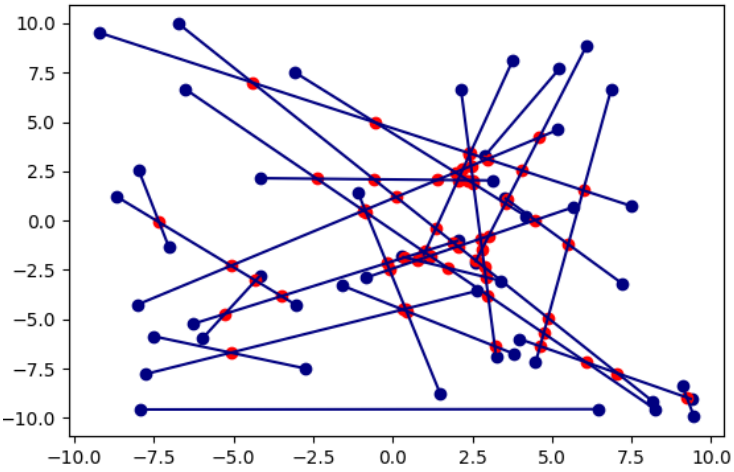
8. Prezentacja wyników

Poniżej przedstawione są grafiki prezentujące wyniki działania algorytmu znajdowania wszystkich przecięć dla różnych zbiorów danych. Sprawdzony jest również przypadek, gdy jeden punkt będący przecięciem jest wielokrotnie znajdowany – taka sytuacja nie sprawiała żadnych problemów dla algorytmu. Nie są prezentowane wyniki działania pierwszego algorytmu (zwracającego informację o istnieniu przecięcia), gdyż jest to jedynie mocno uproszczona wersja docelowego algorytmu zamiatania. Animacje przedstawiające działanie obydwu algorytmów można zobaczyć przy pomocy pliku Jupyter Notebooka załączonego do sprawozdania. Jest tam też dodana możliwość zadania własnego zbioru odcinków, bądź wygenerowania losowych odcinków o zadanych parametrach. Na poniższych grafikach czerwone punkty oznaczają znalezione przecięcia, a niebieskie – początki i końce odcinków. W przypadku animacji (możliwej do wyświetlenia z poziomu pliku z kodem programu) dodatkowo kolorem zielonym oznaczane są odcinki aktualnie będące na miotle i aktualnie rozważany punkt, a kolorem szarym odcinki zdjęte z miotły i przetworzone punkty.

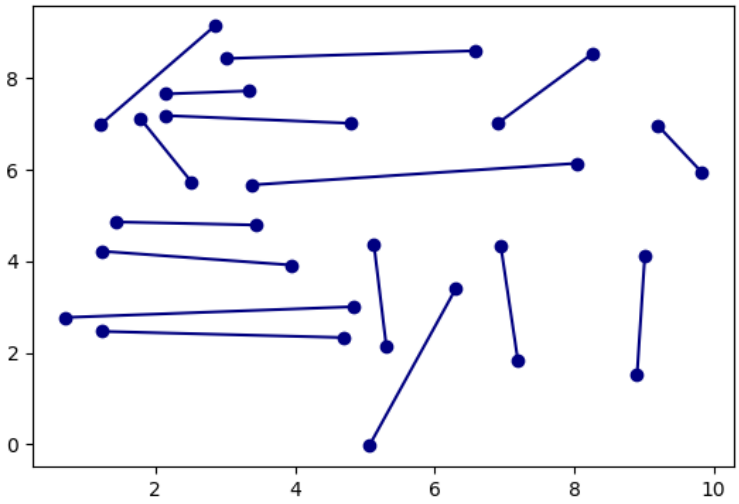
Rys. 1. Zbiór odcinków 1 z przecięciami



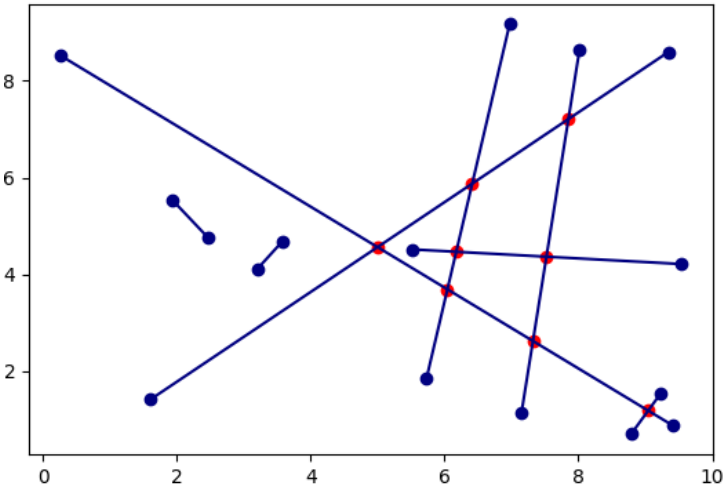
Rys. 2. Zbiór odcinków 2 (wygenerowany losowo) z przecięciami



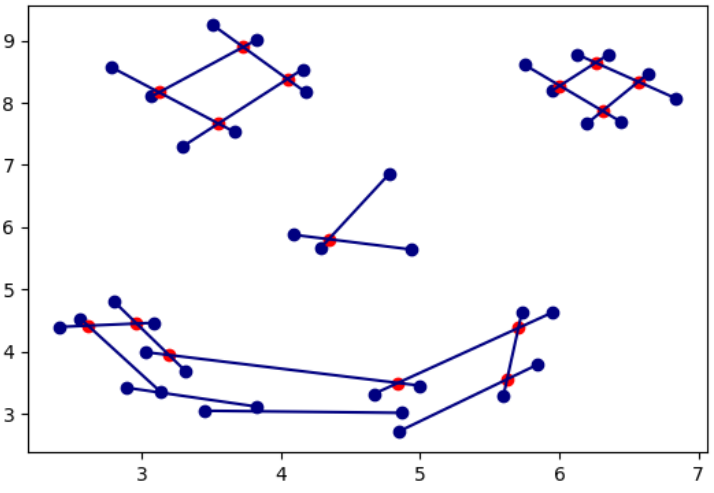
Rys. 3. Zbiór odcinków 3 bez przecięć



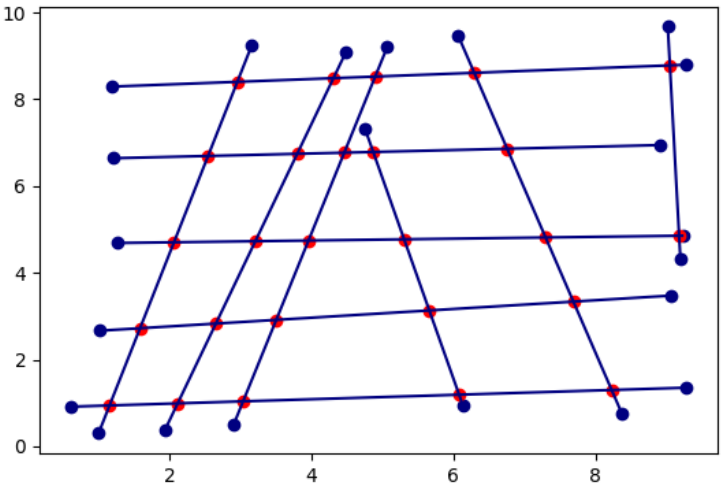
Rys. 4. Zbiór odcinków 4 z przecięciami – sytuacja z kilkukrotnym wykryciem jednego punktu przecięcia



Rys. 5. Zbiór odcinków 5 „Buźka” z przecięciami



Rys. 6. Zbiór odcinków 6 „Kratka” z przecięciami



9. Wnioski

Zaimplementowane w ćwiczeniu algorytmy zwracają oczekiwane wyniki dla przedstawionych zbiorów danych, a także dla losowo generowanych dużych zbiorów danych, gdzie do sprawdzenia poprawności zastosowano porównanie wyników z algorytmem kwadratowym. O ile wykrycie pojedynczego przecięcia nie jest trudnym zadaniem, to poprawne wykrywanie wszystkich punktów przecięcia wymaga rozbudowania algorytmu zamiatania i uwzględnienia w nim sytuacji wyjątkowych – zamiany miejscami punktów w uporządkowanym zbiorze i wielokrotnego znajdowania tego samego punktu przecięcia. Zastosowane metody umożliwiły poprawne radzenie sobie algorytmu z tymi sytuacjami. Pomocne w implementacji algorytmów było stworzenie odpowiednich struktur danych dla punktu i odcinka, dzięki którym możliwe było łatwe uzyskiwanie potrzebnych do działania algorytmu informacji.