Algorytmy geometryczne

Sprawozdanie z ćwiczeń nr. 4
Wojciech Łoboda
Grupa nr. 1 (środa_13_P_1)

Środowisko oraz narzędzia:

- System operacyjny: Windows 10 x64
- Procesor: Intel Core i7-7700HQ 2.80GHz
- Pamięć RAM: 8 GB
- Środowisko: Jupyter Notebook
- Visual Studio Code

Ćwiczenie wykonano przy pomocy języka python3 z wykorzystaniem następujących bibliotek:

- Numpy
- Matplotlib
- sortedcontainers

Opis ćwiczenia

Ćwiczenie polegało na zaimplementowaniu algorytmu zamiatania sprawdzającego czy istnieje przynajmniej jedno przecięcie miedzy zadanymi odcinkami, należało również zaimplementować algorytm wyznaczające wszystkie przecięcia odcinków. Oba algorytmy należało odpowiednio zwizualizować. Odcinki powinny zadawane przy pomocy narzędzia graficznego lub generowane losowo ze współrzędnymi z ustalonego zakresu. Przyjęta tolerancja dla zera wynosepsilon = 10^{-12}

Zestaw danych

Interaktywnie dodane odcinki można pobrać z wykresu dzięki funkcji *get_lines_from_plot(plot)*, funkcja akceptuje jako parametr obiekt plot reprezentujący wykres i zwraca listę odcinków reprezentowanych jako lista z dwoma elementami będącymi współrzędnymi końców odcinka.

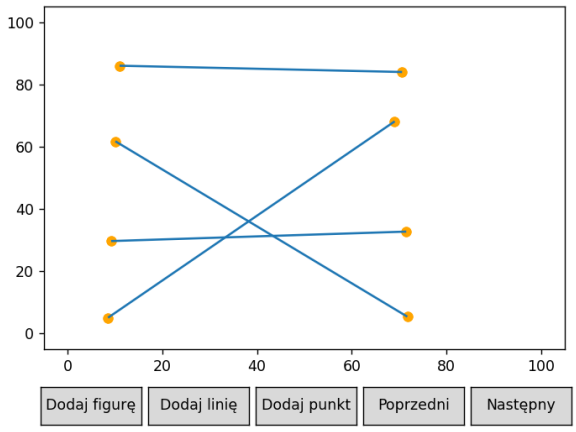
Odpowiednie odcinki można również generować za pomocą funkcji *generate_random_lines(n, bound_x, bound_y)* jako parametry przyjmuje ona liczbę punktów do wygenerowania oraz zakres współrzędnych, gdzie $x\epsilon < 0,\ bound\_x >,\ y\epsilon < 0,\ bound\_y >$.

W zbiorze danych nie może być odcinków pionowych oraz żadna para odcinków nie powinna mieć końców o jednakowej współrzędnej x-owej, zbiory wygenerowane losowo spełniają te założenia.

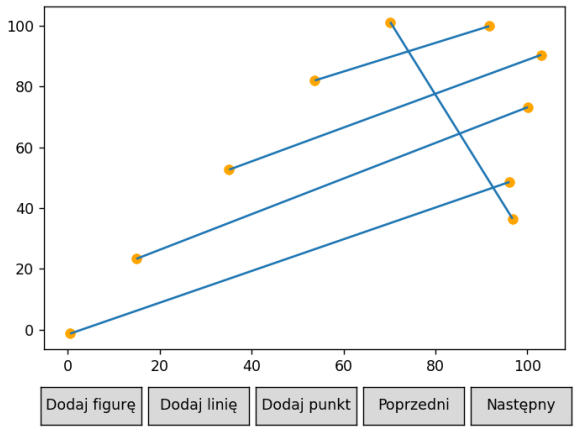
Odcinki można zapisywać do pliku w formacie json przy pomocy funkcji *save_lines_to_file(lines, name),* gdzie *lines* to lista odcinków a *name* to nazwa pliku w formacie json. Odczyt jest możliwy za pomocą funkcji *get_saved_lines(name),* zwracającą listę odcinków zapisaną w pliku o nazwie *name.json*.

Wykorzystane zestawy danych:

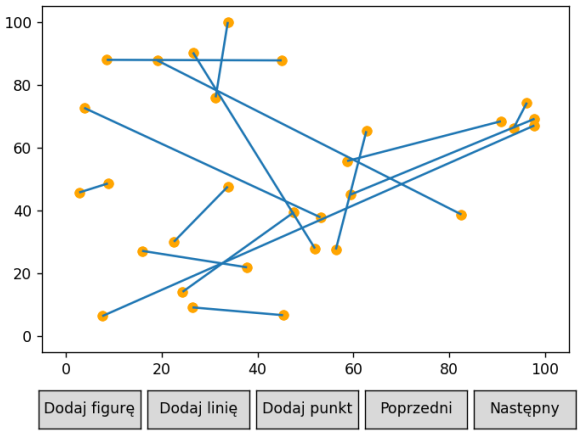
Wykres 1.1 Zestaw A (4 odcinki)



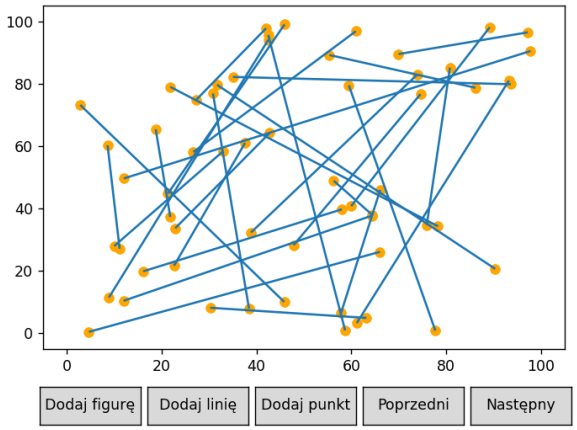
Wykres 1.2 Zestaw B (5 odcinków)



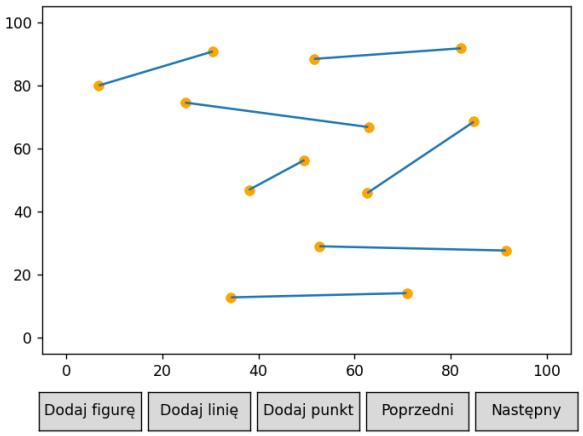
Wykres 1.3 Zestaw C (15 odcinków)



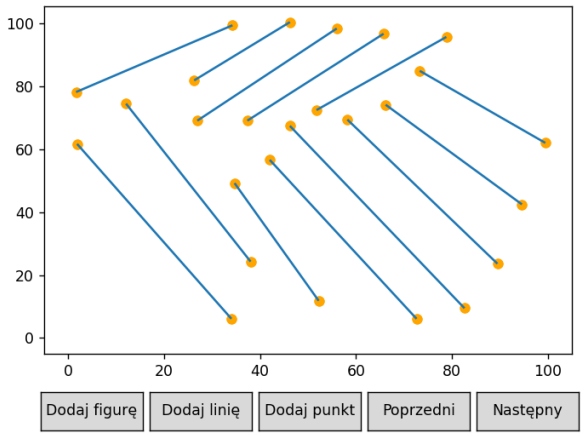
Wykres 1.4 Zestaw D (30 odcinków)



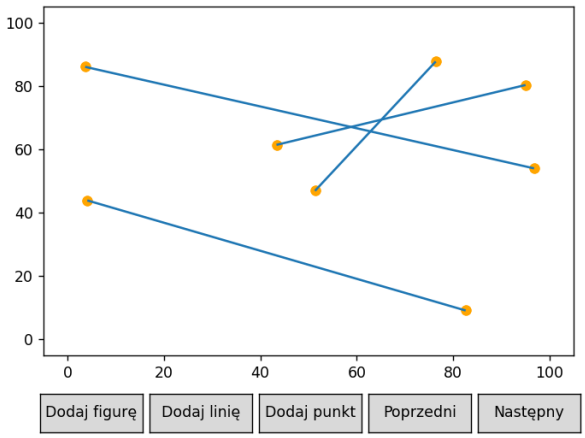
Wykres 1.5 Zestaw E (7 odcinków)



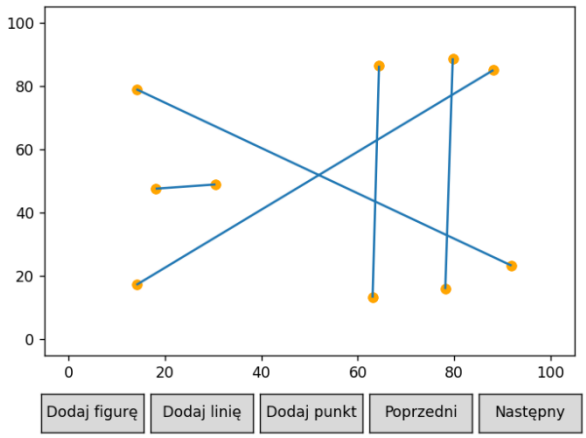
Wykres 1.6 Zestaw F (13 odcinków)



Wykres 1.7 Zestaw G (4 odcinki)



Wykres 1.8 Zestaw H (5 odcinków)



Wykorzystane struktury danych

W obu algorytmach wykorzystane zostały jednakowe struktury danych do przechowywania aktualnego stanu miotły oraz zdarzeń. Struktury oparte są o zbalansowane drzewo binarne implementowane w tym przypadku przez strukturę *SortedSet* należącą do biblioteki *sortedcontainers*. SortedSet umożliwia przechowywanie elementów uporządkowanych zgodnie z zadanym komparatorem. SortedSet pozwala na dodawanie, usuwanie i znajdowanie elementu w złożoności czasowej O(log(n)), gdzie n to rozmiar struktury. Struktura pozwala na przechowywanie jedynie unikalnych elementów.

Struktura stanu to SortedSet zawierający w sobie obiekty klasy *Lines*, Klasa ta definiuje swój komparator pozwalający określić kolejność odcinków przecinających miotłę w jej aktualnej pozycji.

Struktura zdarzeń to również SortedSet ale zawierający obiekty klasy *Events*. Klasa ta reprezentuje zdarzenie i posiada jego typ, współrzędną x oraz indeksy odcinków których dotyczy zdarzenie. Klasa ta definiuje komparator pozwalający na przechowywanie zdarzeń w strukturze w odpowiedniej kolejności ze względu na ich x-ową współrzędną oraz metodę określającą równość zdarzeń jeżeli jest to zdarzenie tego samego typu dotyczące jednakowych odcinków.

Tak dobrana struktura nie pozwala na wielokrotne dodania tego samego przecięcia oraz w prosty sposób umożliwia wszystkie operacje potrzebne do działania algorytmów z odpowiednią złożonością czasową.

Obsługiwanie zdarzeń:

Dodawanie odcinków do struktury sprowadza się do wykorzystania metody *add()* SortedSetu a następnie na sprawdzeniu czy nowo dodany odcinek przecina się ze swoimi sąsiadami w strukturze. Do uzyskania indeksu elementu należącego do struktury wykorzystuję metodę *index()*. Posiadając indeks znajdowani są sąsiedzi odcinka w strukturze i sprawdzane jest ich przecięcie.

Do usuwania odcinków wykorzystywana jest metoda *remove()* usuwająca element SortedSetu po usunięciu odcinka sprawdzane jest ewentualne przecięcie między sąsiadami usuniętego odcinka.

Przecięcie odcinków obsługiwane jest poprzez usunięcie przecinających się odcinków ze struktury a następnie dodanie ich ale ze zmienioną aktualną pozycją miotły o x + epsilon, gdzie epsilon to przyjęta tolerancja na zero. Taki zabieg zamienia miejscami odcinki w strukturze. Należy jeszcze sprawdzić przecięcia z nowymi sąsiadami odcinków

Algorytm sprawdzający czy chociaż jedna para odcinków się przecina

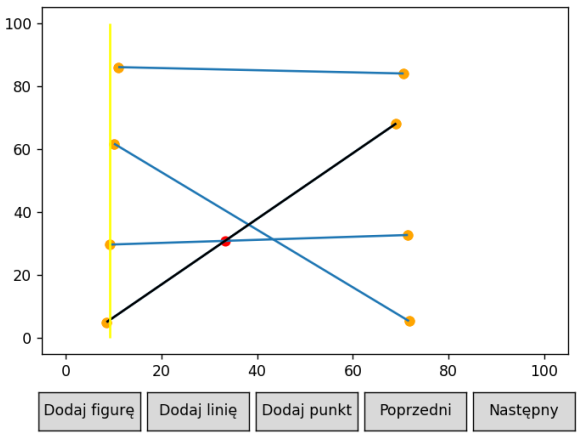
Działanie:

T - Struktura stanu
Q – Struktura zdarzeń

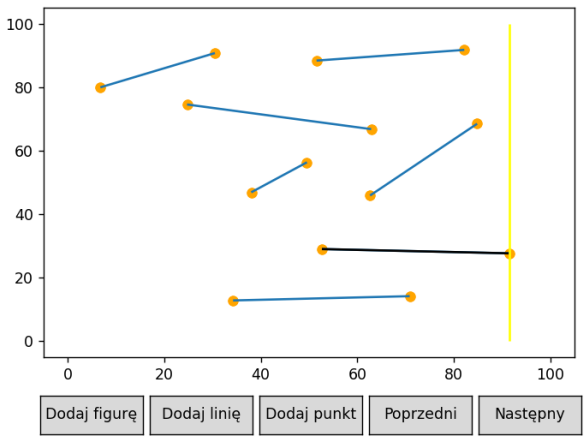
- Do Q należy wstawić zdarzenia reprezentujące końce i początki wszystkich odcinków
- Dopóki w Q znajdują się zdarzenia:
  - pobierz nowe zdarzenie p z Q
  - zaktualizuj T:
    - jeśli p jest lewym końcem odcinka – dodaj odcinek do T
    - jeśli p jest prawym końcem odcinka – usuń odcinek z T
  - zaktualizuj Q:
    - dla każdej pary nowych sąsiadów s i s' z T sprawdź, czy s i s' przecinają się po prawej stronie miotły jeśli tak – zwracane jest True (znaleziono przecięcie).
- Zwracane jest False (nie znaleziono przecięcia)

Wyniki działania algorytmu dla wybranych zbiorów danych (Punkt oznaczony na czerwono to wykryte przecięcie):

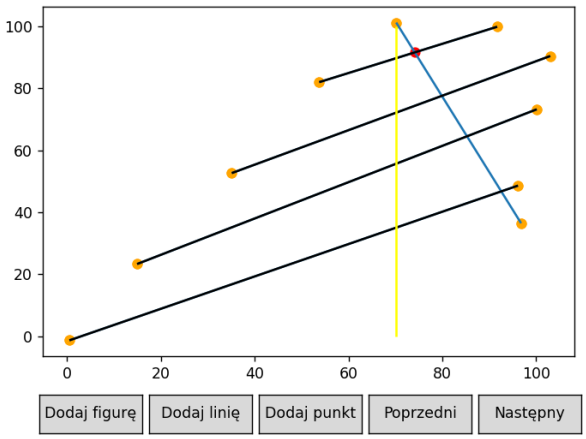
Wykres 2.1 Zestaw A (Znaleziono przecięcie)



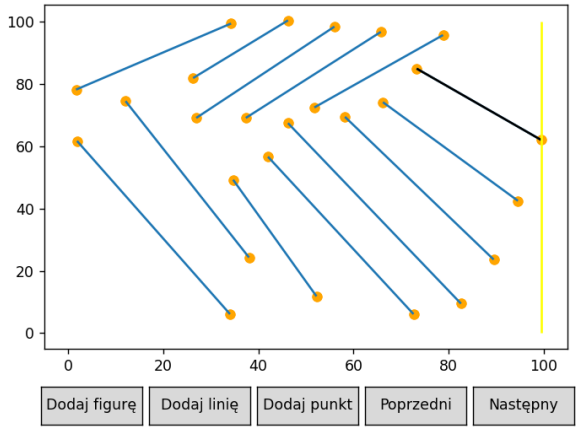
Wykres 2.2 Zestaw E (Bez przecięcia)



Wykres 2.3 Zestaw B (Znaleziono przecięcie)



Wykres 2.4 Zestaw F (Bez przecięcia)



Algorytm wyznaczający wszystkie przecięcia odcinków

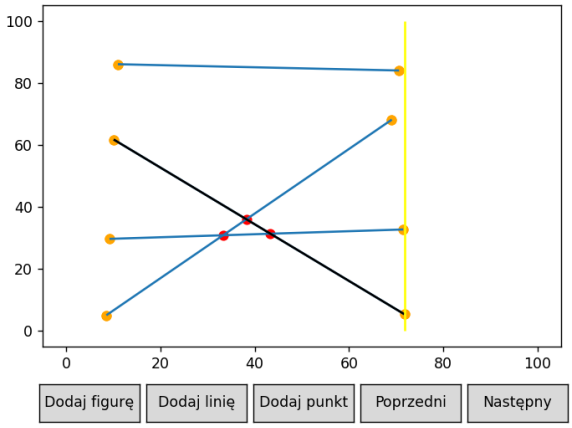
Działanie:
Utwórz pustą strukturę stanu T
Utwórz strukturę zdarzeń Q – wstaw zdarzenia początków i końców odcinków
Powtarzaj

- pobierz nowe zdarzenie p z Q
- zaktualizuj T:
  - jeśli p jest lewym końcem odcinka – dodaj odcinek do T
  - jeśli p jest prawym końcem odcinka – usuń odcinek z T
  - jeśli p jest punktem przecięcia s i s', zmień porządek s i s' w T
- zaktualizuj Q:
  - dla każdej pary nowych sąsiadów s i s' z T sprawdź, czy s i s' przecinają się po prawej stronie miotły jeśli tak – dodaj punkt przecięcia do Q

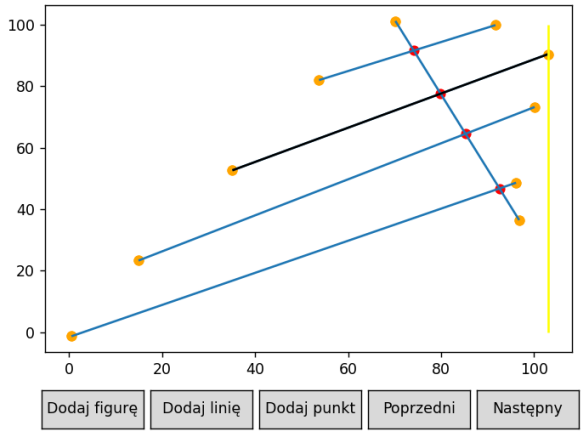
Aż Q będzie puste

Wyniki działania algorytmu dla zbiorów danych (Punkty oznaczone na czerwono to punkty przecięcia):

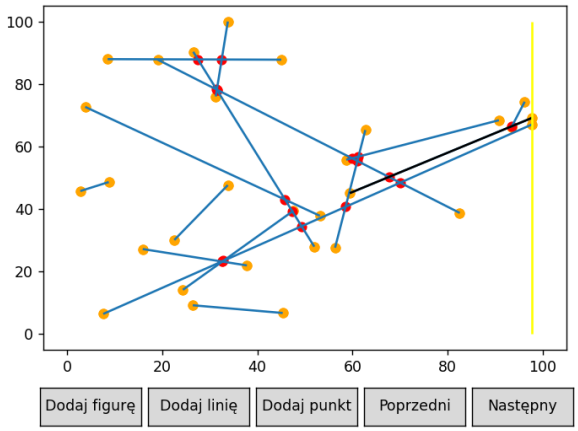
Wykres 3.1 Zestaw A (3 przecięcia)



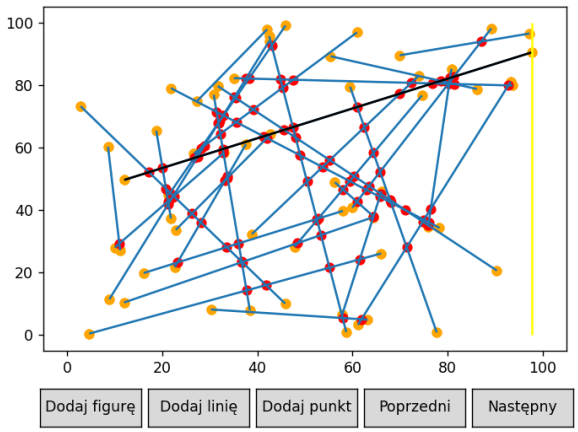
Wykres 3.2 Zestaw B (4 przecięcia)



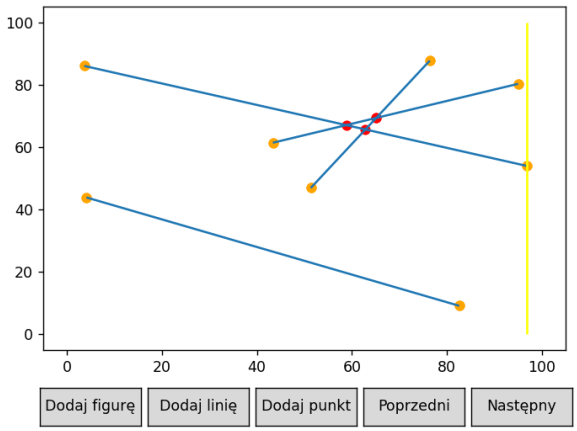
Wykres 3.3 Zestaw C (18 przecięć)



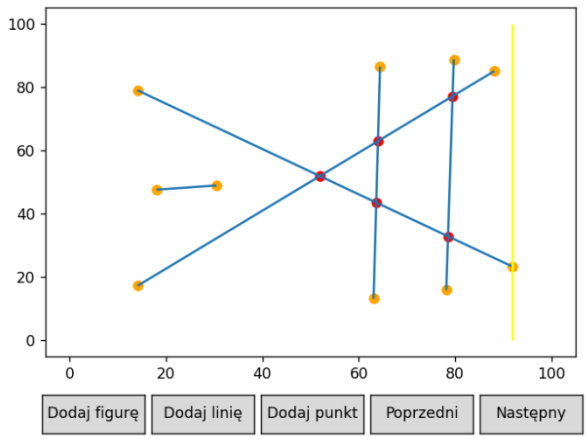
Wykres 3.4 Zestaw D (100 przecięć)



Wykres 3.5 Zestaw G (3 przecięcia)

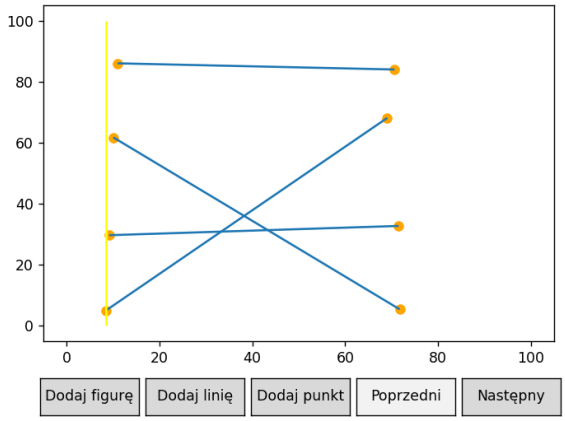


Wykres 3.6 Zestaw H (5 przecięć)

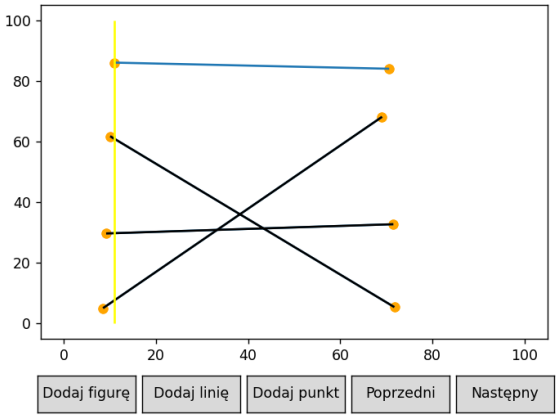


Wizualizacja kroków algorytmu dla zestawu A:

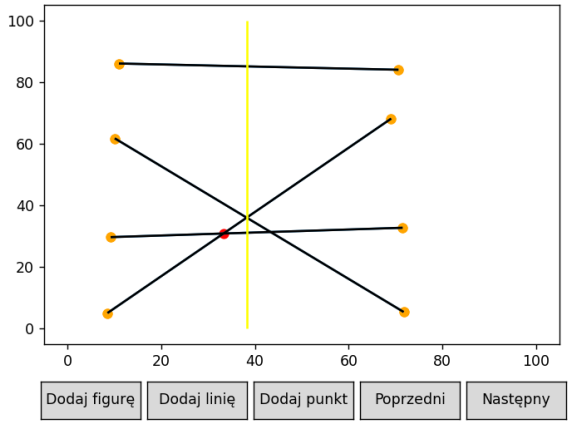
Wykres 4.1 Krok 0



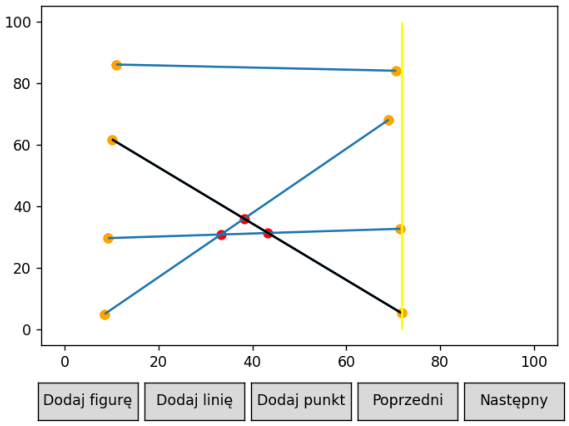
Wykres 4.2 Krok 3



Wykres 4.3 Krok 6



Wykres 4.4 Krok 12



Wnioski

Zaimplementowane algorytmy znajdują poprawne przecięcia dla wszystkich testowanych zestawów danych. Każde przecięcie znajdowane jest jednokrotnie dzięki wykorzystanej strukturze danych. Nie była konieczna zmiana struktury między wykorzystanymi algorytmami. W algorytmie sprawdzania czy choć jedna para odcinków się przecina strukturą stanu mogła być bardziej uproszczona niż ta zastosowana, ponieważ po wykryciu jakiegokolwiek przecięcia przerywamy algorytm i nigdy nie zamieniamy kolejności odcinków należących do stanu miotły.