



# Obliczanie grafu widoczności

Algorytmy geometryczne  
Projekt

Norbert Morawski  
Dariusz Piwowarski

8 stycznia 2022

## Spis treści

<b>1 Informacje techniczne</b>	<b>3</b>
1.1 Platforma . . . . .	3
1.2 Użyte biblioteki i narzędzia . . . . .	3
1.3 Opis programu . . . . .	3
1.4 Moduły . . . . .	3
<b>2 Informacje użytkownika</b>	<b>5</b>
2.1 Uruchomienie programu . . . . .	5
2.2 Funkcje programu . . . . .	6
2.2.1 Sekcja 1 – Przykład 1 - Wyznaczenie grafu widoczności . . . . .	6
2.2.2 Sekcja 2 – Wyznaczanie grafu widoczności dla wprowadzonych figur . . . . .	6
2.2.3 Sekcja 3 – Przykład 2 - Wyznaczanie najkrótszej ścieżki pomiędzy dwoma punktami na płaszczyźnie z przeszkodami . . . . .	6
2.2.4 Sekcja 4 – Wyznaczanie najkrótszej ścieżki pomiędzy dwoma punktami na płaszczyźnie z przeszkodami, na podstawie wprowadzonych wprowadzonych figur i punktów . . . . .	7
2.2.5 Sekcja 5 – Testy . . . . .	7
2.2.6 Sekcja 6 – Manualne wprowadzanie figur i punktów . . . . .	7
<b>3 Sprawozdanie</b>	<b>8</b>
3.1 Opis problemu . . . . .	8
3.2 Algorytm wyznaczania widocznych wierzchołków dla punktu $p$ . . . . .	8
3.3 Struktura zdarzeń . . . . .	9
3.4 Struktura stanu . . . . .	9
3.5 Sprawdzanie widoczności . . . . .	10
3.6 Kroki działania algorytmu . . . . .	16
3.7 Reprezentacja grafu . . . . .	21
3.8 Testy . . . . .	23
3.9 Testy losowe . . . . .	32
3.9.1 Czasy wykonania dla testów losowych . . . . .	37
3.10 Testy znajdowania najkrótszej ścieżki . . . . .	39
<b>4 Wnioski</b>	<b>41</b>
<b>5 Bibliografia</b>	<b>41</b>

# 1 Informacje techniczne

## 1.1 Platforma

System	Arch Linux/Windows 10
Architektura	x86_64
Język programowania	Python
Wersja interpretera	3.9.7

## 1.2 Użyte biblioteki i narzędzia

NumPy	1.21.2
Matplotlib	3.4.3
Jupyter Notebook	6.4.4

## 1.3 Opis programu

Program został napisany w języku Python. Interfejs dla użytkownika dostępny jest w pliku .ipynb, wykorzystuje on narzędzie Jupyter Notebook.

## 1.4 Moduły

Poniżej znajduje się opis modułów i metod w nich występujących.

**helpers.py** Zawiera funkcje pomocnicze do obliczania grafu widoczności.

- `dist_p_to_intersection(p, w, line)` – Oblicza dla zadanych punktów  $p$ ,  $w$  i pewnego odcinka, odległość punktu  $p$  od przecięcia  $pw$  z tym odcinkiem. Jeśli są równoległe zwracany jest środek zadanego odcinka,
- `intersection(line1, line2)` – Zwraca *True/False* w zależności, czy zadane odcinki się przecinają,
- `orient(a, b, c)` – Określa w jaki sposób zorientowane są trzy zadane punkty. Zwraca wartość 1, gdy są zorientowane przeciwnie do ruchu wskazówek zegara; 0, gdy są współliniowe; -1 gdy ich orientacja jest zgodna z ruchem wskazówek zegara,
- `dist(a, b)` – Oblicza odległość pomiędzy dwoma punktami na płaszczyźnie.

**create\_visibility\_graph.py** Zawiera implementację algorytmu obliczania grafu widzialności.

- `find_internal_lines` – Implementuje algorytm, który znajduje dla każdego punktu figury, inne wierzchołki tego samego wielokąta, takie że linia je łącząca jest skierowana do wnętrza figury. Złożoność  $O(n^2)$ ,
- `create_visibility_graph` – Tworzy graf widzialności.

**VisibilityGraph.py** Klasa zawierająca reprezentację finalnego grafu widzialności.

- `__init__` – Konstruktor,
- `add_edge` – Dodaje krawędź do grafu,
- `get_graph` – Zwraca graf reprezentowany w postaci opisywanej w punkcie 3.7,
- `get_points` – Zwraca listę krotek reprezentujących zbiór wierzchołków grafu,
- `get_lines` – Zwraca listę krawędzi,
- `get_lines_separately` – Zwraca listę list krawędzi z podziałem na wierzchołki początkowe krawędzi,
- `__repr__` – Zwraca reprezentację tekstową klasy.

**Sweeper.py** Klasa zawierająca aktualny stan miotły (punkt początkowy i końcowy)

**shortest\_path.py** Implementacja algorytmu Dijkstry, który wyznacza najkrótszą ścieżkę pomiędzy przeszkodami.

- `dijkstra` – Implementuje algorytm Dijkstry. Złożoność  $O(n \cdot \log(n))$ ,
- `shortest_path` – Funkcja wyznaczająca najkrótszą ścieżkę pomiędzy dwoma zadanimi punktami, na płaszczyźnie zawierającej przeszkody w postaci wielokątów.

**random\_figures\_generator.py** Zawiera funkcje służące do generowania w sposób losowy figur na płaszczyźnie.

- `generate_random_figures` – Funkcja zwraca listę  $n$  losowo wygenerowanych kwadratów, dla zadanego  $n$  oraz rozmiaru płaszczyzny generowania.

**Point.py** Klasa zawierająca reprezentację punktu na płaszczyźnie.

- `__init__` – Konstruktor,
- `with_line` – Dodaje krawędź incydentną z tym wierzchołkiem,
- `has_line_to` – Sprawdza czy wierzchołek jest incydentny z podaną krawędzią,
- `has_internal_line_to` – Sprawdza czy krawędź pomiędzy tym a podanym wierzchołkiem jest skierowana do wnętrza figury,
- `__repr__` – Zwraca reprezentację tekstową klasy.

**Line.py** Klasa zawierająca reprezentację odcinka na płaszczyźnie. Wewnętrznie przechowuje jego początek i koniec.

- `__init__` – Konstruktor,
- `__repr__` – Zwraca reprezentację tekstową klasy,
- `__eq__` – Sprawdza czy linie są sobie równe (porównuje czy mają takie same końce i początki),
- `__gt__` – Sprawdza czy linia następuje po podanej linii w strukturze stanu.

**Figure.py** Klasa zawierająca reprezentację figury na płaszczyźnie. Wewnętrznie przechowuje listę punktów należących do figury, w kolejności przeciwej do ruchu wskazówek zegara.

- `__init__` – Konstruktor,
- `__repr__` – Zwraca reprezentację tekstową klasy.

**plotter/Plotter.py** Klasa obsługująca generowanie wykresów krok-po-kroku.

- `__init__` – Konstruktor,
- `init_limits` – Pobiera maksymalne i minimalne współrzędne punktów w grafie,
- `new_partial_plot` – Tworzy nowy częściowy wykres,
- `partial_plot` – Rysuje nowy częściowy wykres,
- `sum_up` – Rysuje podsumowanie jednej iteracji zewnętrznej pętli algorytmu.

**plotter/SequencePlotter.py** Klasa pomocnicza obsługująca generowanie osobnych wykresów w osobnych plikach, za każdym razem inkrementując numer wykresu.

- `__init__` – Konstruktor,
- `next` – Tworzy nowy wykres.

**plotter/Plot.py** Klasa pomocnicza zawierająca podstawowe funkcje obsługi wykresów.

**plot\_tool.py** Dostarczone narzędzie graficzne. Została dopisana funkcjonalność opisywania wierzchołków kolejnymi literami.

### plot\_tool\_helpers.py

- `visibility_graph_scenes` – Zwraca utworzoną sceny dla wykresu grafu widoczności,
- `visibility_graph_scenes_separately` – Zwraca utworzoną sceny dla wykresu grafu widoczności, z podziałem na kolejne wierzchołki,
- `shortest_path_scene` – Zwraca utworzoną scenę dla wykresu najkrótszej ścieżki,
- `get_figures_from_plot` – Zwraca figury dodane do wykresu jako obiekty klasy *Figure*,
- `get_points_from_plot` – Zwraca punkty dodane do wykresu jako obiekty klasy *Point*.

## 2 Informacje użytkownika

### 2.1 Uruchomienie programu

Należy w katalogu z plikiem `project.ipynb` uruchomić narzędzie Jupyter Notebook, a następnie wybrać z listy ten plik i go uruchomić.

## 2.2 Funkcje programu

Aby wszystkie funkcje działały prawidłowo, w pierwszej kolejności należy uruchomić komórkę **Konfiguracja**, która importuje potrzebne moduły oraz komórkę **Funkcje pomocnicze do tworzenia rysunków**, która importuje funkcje tworzące sceny dla wykresów.

### 2.2.1 Sekcja 1 – Przykład 1 - Wyznaczenie grafu widoczności

Sekcja ta zawiera prosty przykład wyznaczania grafu widoczności. Na początku należy uruchomić komórkę **Wczytywanie przykładowych figur**. Kiedy figury zostaną załadowane można uruchomić następujące komórki:

- **Wyświetlenie grafu widoczności** – Po uruchomieniu komórki wyświetlany jest graf widoczności. Niebieskimi liniami zaznaczone są jego krawędzie, a wierzchołki są oznaczone zielonymi punktami i są nazwane kolejnymi literami alfabetu. Używając przycisków **Poprzedni**, **Następny** można zmienić scenę. W tym przypadku na drugiej scenie znajduje się graf widoczności z zaznaczonymi poprzez czerwone przerywane linie wprowadzonymi figurami.
- **Wyświetlenie incydentnych krawędzi grafu widoczności dla kolejnych wierzchołków** – Działa podobnie co poprzednia komórka, z tą różnicą, że wyświetlany graf posiada krawędzie wychodzące z jednego wierzchołka. Aby zmienić ten wierzchołek, należy przejść do kolejnej sceny.
- **Zapisywanie kolejnych kroków algorytmu do plików .jpg w folderze out** – Tak jak wskazuje opis komórki, zapisuje ona kolejne kroki algorytmu w postaci rysunków do folderu *out*. Objasnienie co reprezentują kolejne kroki znajduje się w punkcie 3.6.

### 2.2.2 Sekcja 2 – Wyznaczanie grafu widoczności dla wprowadzonych figur

Sekcja ta działa podobnie jak poprzednia, różnią się tylko pierwszą komórką. W tym przypadku jest nią **Wprowadzanie figury**. Wyświetla ona pusty wykres, na którym po kliknięciu **Dodaj figurę**, dodajemy kolejne wielokąty używając lewego przycisku myszy. Aby zakończyć dodawanie należy zamknąć figurę, klikając na punkt dodany jako pierwszy. Ważne jest, że wierzchołki figury muszą być wprowadzane w kolejności odwrotnej do ruchu wskazówek zegara.

Pozostałe komórki działają analogicznie jak w poprzedniej sekcji.

### 2.2.3 Sekcja 3 – Przykład 2 - Wyznaczanie najkrótszej ścieżki pomiędzy dwoma punktami na płaszczyźnie z przeszkodami

Sekcja zawiera przykład wyznaczania najkrótszej ścieżki na płaszczyźnie z przeszkodami w postaci wielokątów. Podobnie jak w sekcji 1 najpierw należy uruchomić komórkę **Wczytywanie przykładowych figur i punktów**. Druga i ostatnia w tej sekcji komórka, opisana jako **Wyświetlanie najkrótszej ścieżki**, wyświetla rysunek, na którym niebieskimi liniami zaznaczona jest najkrótsza ścieżka pomiędzy dwoma zadanymi punktami *A* i *B*. Przeszkody w postaci figur oznaczone są czerwoną przerywaną linią.

#### 2.2.4 Sekcja 4 – Wyznaczanie najkrótszej ścieżki pomiędzy dwoma punktami na płaszczyźnie z przeszkodami, na podstawie wprowadzonych wprowadzonych figur i punktów

Sekcja różni się od poprzedniej pierwszą komórką, tutaj jest nią **Wprowadzanie figury**. Pozwala ona na dodawanie figur przez użytkownika podobnie jak w sekcji 2, z tą różnicą, że konieczne jest także dodanie dwóch punktów. Aby to zrobić klikamy jednokrotnie przycisk **Dodaj punkt**, a następnie wybieramy lewym przyciskiem myszy dwa punkty na wykresie. Należy pamiętać, że wierzchołki figury muszą być wprowadzane w kolejności odwrotnej do ruchu wskazówek zegara.

#### 2.2.5 Sekcja 5 – Testy

Zawiera serię przygotowanych danych testowych. Dane te są ładowane w komórce **Załadowanie testów**, a następnie mamy możliwość ich wykonania i zapisania wyników w postaci wykresów w komórce **Wykonanie testów i zapisanie rysunków wynikowych w folderze out** oraz sprawdzenie czasu wykonania algorytmu dla danego testu w komórce **Wyświetlanie czasu wykonania poszczególnych testów**.

#### 2.2.6 Sekcja 6 – Manualne wprowadzanie figur i punktów

Sekcja pozwala na ręczne wprowadzenie figur i punktów, na podstawie których utworzony zostanie graf widoczności oraz wyznaczona zostanie najkrótsza ścieżka, jeśli podane będą dwa dodatkowe punkty. Dane te wprowadzamy w komórce **Wczytywanie figur i punktów** w następujący sposób:

- **Figury** – Dodajemy je w postaci obiektów klasy *Figure* do listy *F*. Obiekt klasy *Figure* tworzymy podając do konstruktora listę obiektów klasy *Point*. Obiekt klasy *Point* tworzymy z kolei podając do konstruktora dwuelementową krotkę reprezentującą współrzędne punktu.
- **Punkty** – Podajemy je jako listę *P* obiektów klasy *Point* tworzonych w sposób opisany powyżej. Lista powinna zawierać 2 punkty, jeśli będzie ich za mało najkrótsza ścieżka nie zostanie wyznaczona, natomiast jeśli będzie ich za dużo, wybrane zostaną 2 pierwsze punkty z listy.

```
F = [Figure([Point(0, 0), Point(1, 0), Point(0, 1)]),
      Figure([Point(3, 3), Point(3, 2), Point(4, 2)])]

P = [Point(-2, -2), Point(4, 3)]
```

Powyższy fragment kody przedstawia przykład wprowadzania opisywanych wyżej figur i punktów.

## 3 Sprawozdanie

### 3.1 Opis problemu

**Graf widoczności** Jest to graf gdzie zbiorem wierzchołków jest zbiór punktów na płaszczyźnie, a krawędzie istnieją tylko kiedy wierzchołki "widzą się", tzn. nie istnieje pomiędzy nimi przeszkoda w postaci figury.

**Zastosowanie** Jednym z zastosowań grafu widoczności jest możliwość wyznaczenia na jego podstawie najkrótszej ścieżki pomiędzy dwoma punktami na płaszczyźnie, na której znajdują się przeszkody w postaci wielokątów. Aby to zrobić wystarczy nadać krawędziom wagi odpowiadające odległościom pomiędzy punktami na płaszczyźnie, a następnie wykonać na takim grafie dowolny algorytm szukania najkrótszej ścieżki.

**Wstęp** Trywialny algorytm wymagałby sprawdzenia każdej pary wierzchołków w czasie  $O(n^2)$ . Każde takie sprawdzenie wymagałoby przejrzenia wszystkich innych wierzchołków w czasie  $O(n)$ , co prowadziłoby do algorytmu o złożoności  $O(n^3)$ . Jednak jeżeli wykorzystamy zrównoważone drzewo binarne do wyszukiwania przeskód, jesteśmy w stanie obniżyć złożoność do poziomu  $O(n^2 \cdot \log(n))$ .

**Algorytm** Problem sprowadza się do wyznaczenia widzianych wierzchołków dla każdego z  $n$  punktów. Do tego celu wykorzystamy algorytm zamiatania. Płaszczyzna jest zamiatana poprzez obrót miotły zaczepionej w rozważanym punkcie, zgodny z ruchem wskazówek zegara.

**Zdarzenia i stan miotły** Zdarzeniami będą punkty na płaszczyźnie, a struktura stanu będzie przechowywać aktualnie przecinane odcinki w kolejność od najbliższego punktowi zaczepienia do najdalszego.

### 3.2 Algorytm wyznaczania widocznych wierzchołków dla punktu $p$

Algorytm polega na zamiataniu płaszczyzny i odbywa się wg następujących kroków:

1. Posortuj wierzchołki na podstawie kąta (zgodnego z ruchem wskazówek zegara) jaki półprosta z  $p$  do danego wierzchołka tworzy z dodatnią półosią OX. Gdy kąt jest taki sam, wierzchołek bliższy  $p$  powinien znajdować się przed wierzchołkiem dalszym. Wierzchołki w tej kolejności umieść w strukturze zdarzeń.
2. Zaczep miotłę w badanym punkcie i zwróć ją w kierunku dodatniej półosi OX.
3. Dodaj do struktury stanu krawędzie przecinające miotłę w położeniu początkowym.
4. Dla kolejnych punktów  $w$  z struktury zdarzeń:
  - Ustaw miotłę jako półprostą z  $p$  do  $w$ .

- Sprawdź czy punkt  $w$  jest widoczny, jeśli tak dodaj odpowiednią krawędź do grafu widoczności.
- Zaktualizuj strukturę stanu dodając do niej odcinki incydentne leżące po stronie zgodnej z ruchem wskazówek zegara względem miotły i usuwając te leżące po stronie przeciwej do ruchu wskazówek zegara.

### 3.3 Struktura zdarzeń

Struktura zdarzeń została zaimplementowana przy użyciu listy.

**Opis sortowania** Zbiór punktów najpierw dzielony jest na dwa podzbiory względem tego czy dany punkt leży poniżej czy powyżej aktualnie rozpatrywanego punktu  $p$  (w przypadku gdy leżą one na tej samej wysokości co punkt  $p$  podział następuje po wartości współrzędnej  $x$ ). Następnie każda część jest sortowana osobno wbudowaną funkcją *sort* (sortowanie stabilne) dwukrotnie. W pierwszym sortowaniu kluczem jest odległość od punktu zaczepienia, w drugim stosowany jest wyznacznik aby posortować punkty zgodnie z ruchem wskazówek zegara.

Ostatecznie otrzymujemy posortowaną strukturę zdarzeń gdzie punkty występują w kolejności zgodnie z ruchem wskazówek zegara, a punkty wspólniowe ułożone są od najbliższych do najdalszych.

**Złożoność** Struktura zdarzeń nie jest modyfikowana w trakcie działania algorytmu, stąd jej wkład do złożoności jest ograniczony przez sortowanie i wynosi  $O(n \cdot \log(n))$ .

### 3.4 Struktura stanu

Struktura stanu przechowuje krawędzie. Wykorzystuje posortowaną listę (opartą na drzewie binarnym) `SortedList` z biblioteki `sortedcontainers`. Gwarantuje ona czas operacji rzędu  $O(\log(n))$ .

**Opis działania** Gdy miotła natrafia na punkt, brane są pod uwagę krawędzie incydentne z nim. Jeżeli krawędź została już zamieciona, jest ona usuwana ze struktury. Gdy nie została jeszcze zamieciona, jest dodawana do niej. Test po której stronie znajduje się krawędź jest przeprowadzany przy użyciu wyznacznika.

**Porządek elementów** Elementy są utrzymywane w porządku rosnącym względem ich odległości od punktu zaczepienia miotły. Jeżeli oba odcinki zaczynają lub kończą się w tym samym punkcie używamy wyznacznika.

**Złożoność** Na strukturze stanu wykonamy maksymalnie wykonamy  $n$  operacji wstawiania i usuwania, co daje nam czas działania rzędu  $O(n \cdot \log(n))$ .

### 3.5 Sprawdzanie widoczności

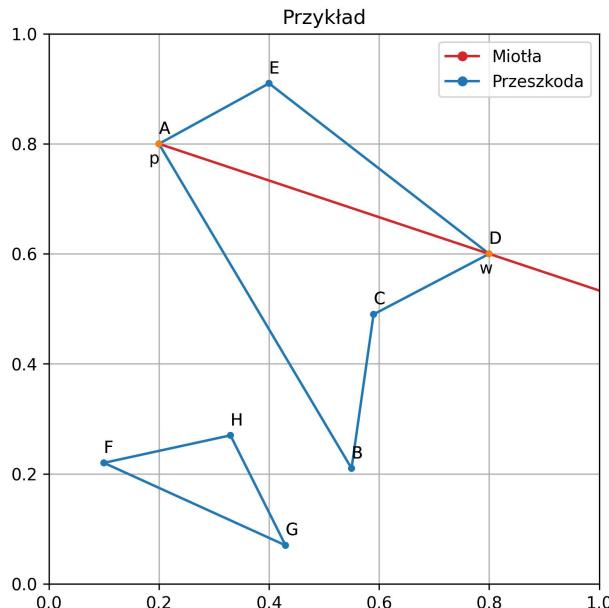
Zgodnie z opisem algorytmu w punkcie 3.2 sprawdzanie widoczności będziemy wykonywać dla zadanych punktów  $p$  i  $w$ . W celu określenia, czy punkt  $w$  jest widoczny będziemy analizować strukturę stanu miotły. W niektórych przypadkach potrzebujemy dodatkowe informacje o punkcie z struktury zdarzeń, który był rozważany poprzednio. Oznaczamy ten punkt jako  $prev\_w$ .

Gdy zaczynamy rozpatrywać punkt, testujemy czy jest on widziany z punktu zaczepienia miotły. W algorytmie spotykamy następujące przypadki:

**Przypadek 1** Punkty  $p$  i  $w$  należą do tej samej figury i wektor  $\overrightarrow{wp}$  jest skierowany do wnętrza wielokąta.

W tym przypadku punkt  $w$  nie jest widoczny, ponieważ linia łącząca  $p$  i  $w$  przechodziłaby przez wnętrze figury.

Przypadek musimy sprawdzić jako pierwszy, ponieważ taka sytuacja spełnia również przypadek 2 lub 3 i wtedy punkt  $w$  zostałby błędnie oznaczony jako widoczny.

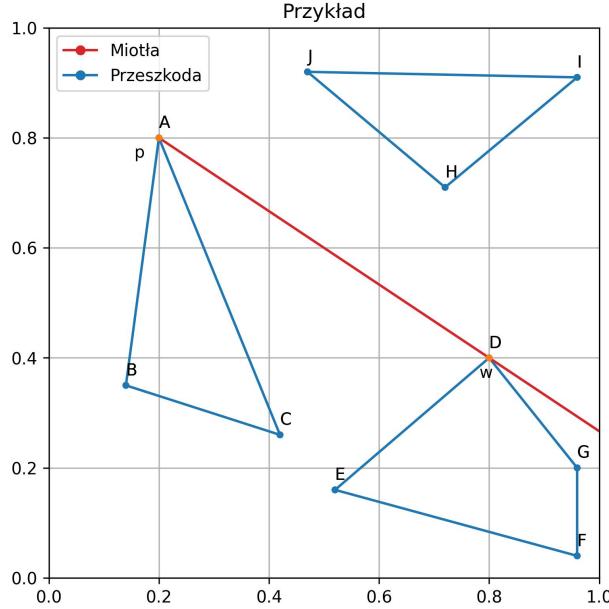


Rysunek 1: Przykład dla przypadku 1

Jak widać na przykładzie wektor  $\overrightarrow{wp}$  jest skierowany do wnętrza wielokąta, zatem pomimo że miotła nie jest przecinana przez żaden odcinek punkt  $w$  nie jest widoczny.

**Przypadek 2** Nie zaszedł przypadek 1 i struktura stanu miotły jest pusta.

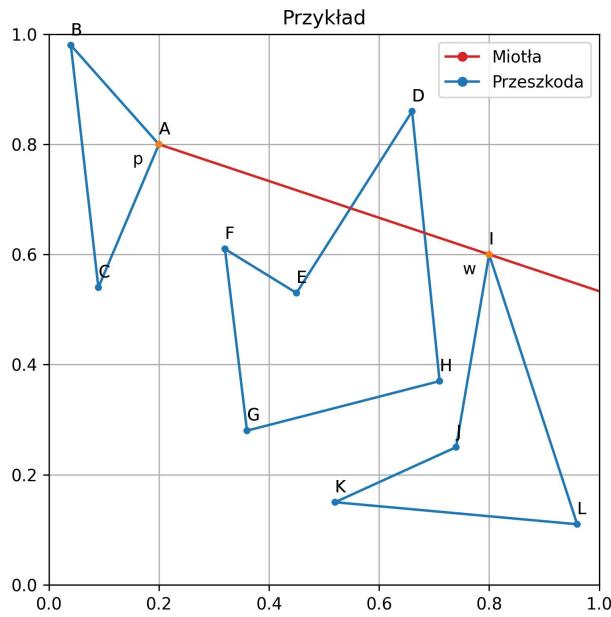
Struktura stanu jest pusta zatem na linii z  $p$  do  $w$  nie może pojawić się żadna przeszkoda. Punkt  $w$  jest zatem widoczny.



Rysunek 2: Przykład dla przypadku 2

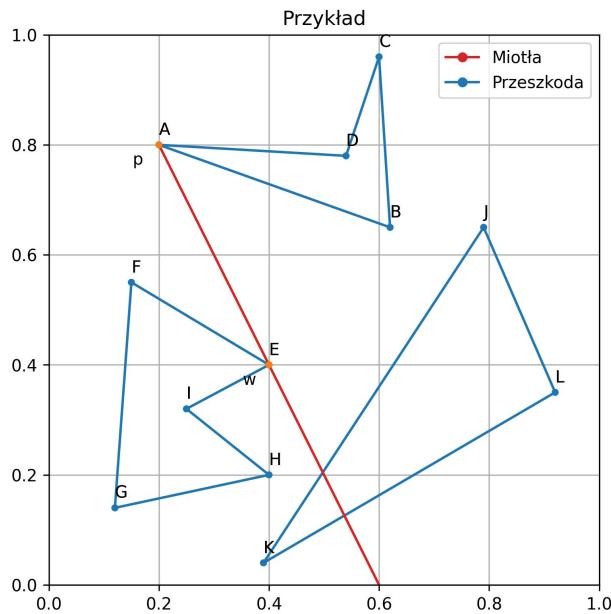
**Przypadek 3** Nie zaszedł żaden z powyższych przypadków i punkty  $p$ ,  $prev\_w$ ,  $w$  nie są współliniowe:

Jest to przypadek, który standardowo będzie występował najczęściej. Aby stwierdzić czy punkt  $w$  jest widoczny należy sprawdzić, czy punkt przecięcia odcinka znajdującego się na samym początku struktury stanu (czyli najbliżej  $p$ ) z miotłą leży bliżej, czy dalej od  $p$  niż  $w$ . Inaczej mówiąc sprawdzamy, czy odcinek  $pw$  przecina odcinek z struktury stanu. Jeżeli tak jest to oznacza, że pomiędzy  $p$  i  $w$  znajduje się przeszkoda i punkt  $w$  jest niewidoczny. W przeciwnym wypadku punkt ten jest widoczny.



Rysunek 3: Przykład 1 dla przypadku 3

Pierwszy odcinek z struktury stanu to  $DE$ , przecina się on z odcinkiem  $pw$ , zatem punkt  $w$  nie jest widoczny.

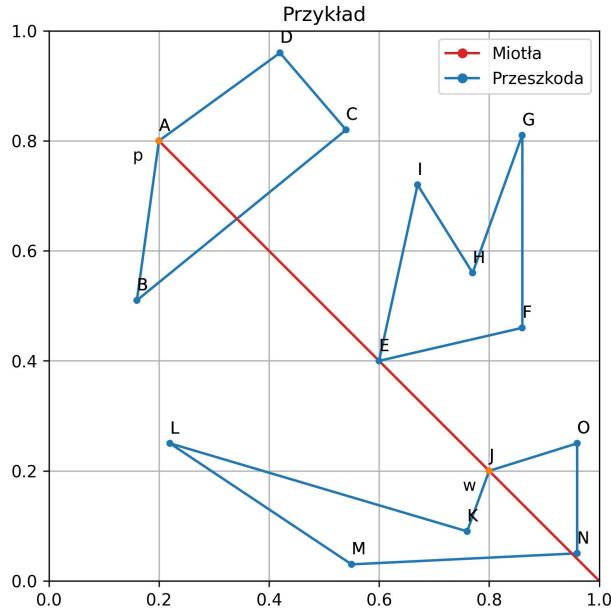


Rysunek 4: Przykład 2 dla przypadku 3

Pierwszy odcinek z struktury stanu to  $KJ$ , nie przecina się on z odcinkiem  $pw$ , zatem punkt  $w$  jest widoczny.

**Przypadek 4** Nie zaszedł żaden z powyższych przypadków, punkty  $p$ ,  $prev\_w$ ,  $w$  są współliniowe oraz punkt  $prev\_w$  nie był widoczny:

Współliniowe punkty w strukturze zdarzeń, są ustawione w kolejności rosnącej odległości od  $p$ , zatem jeżeli  $prev\_w$  był niewidoczny, to  $w$  także nie jest widoczny.



Rysunek 5: Przykład dla przypadku 4

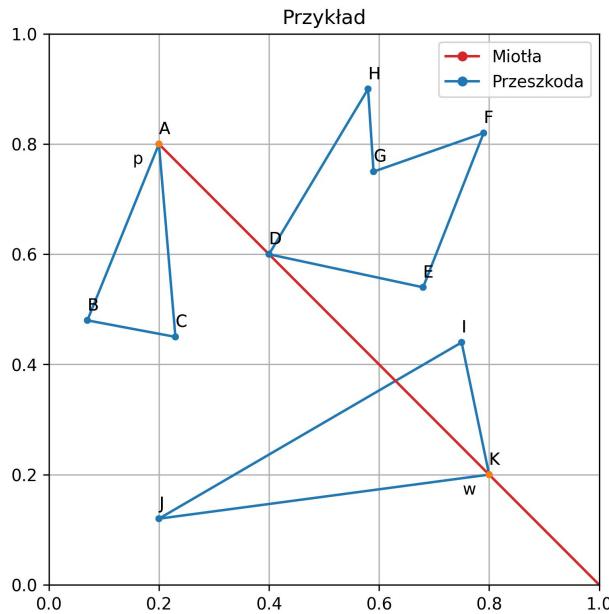
Punkt  $E$  był rozważany bezpośrednio przed punktem  $w$  i jest on niewidoczny. Dodatkowo punkty  $p$ ,  $E$ ,  $w$  są współliniowe. Wynika z tego, że punkt  $w$  również nie jest widoczny.

**Przypadek 5** Nie zaszedł żaden z powyższych przypadków, punkty  $p$ ,  $prev\_w$ ,  $w$  są współliniowe oraz punkt  $prev\_w$  był widoczny:

Przypadek ten jest podobny do przypadku 3, jednak musimy rozważyć go osobno, ponieważ po przetworzeniu punktu  $prev\_w$  na początku struktury stanu mogły zostać umieszczone odcinki, które nie przecinają miotły. Aby w tym przypadku stwierdzić, czy punkt  $w$  jest widoczny sprawdzamy, czy w strukturze stanu znajduje się odcinek, który przecina  $w$   $prev\_w$ .

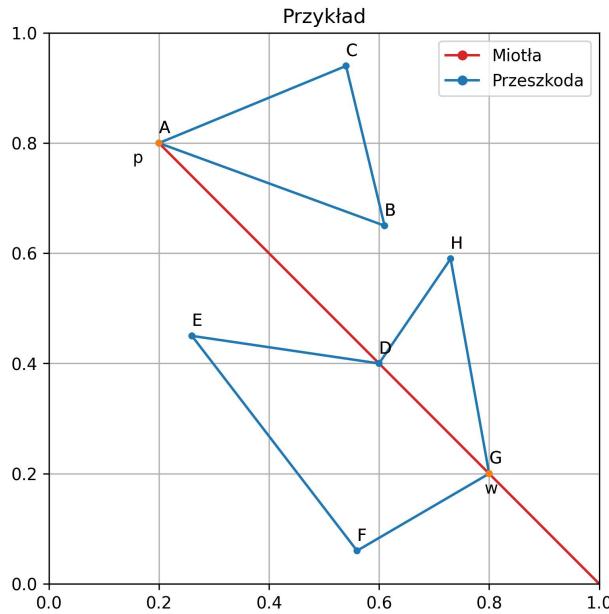
Gdy nie znajdziemy w strukturze stanu takiego odcinka, dodatkowo musimy rozważyć jeszcze jeden szczególny pod-przypadek, podobny do przypadku 1. Zachodzi on, gdy  $prev\_w$  i  $w$  należą do tej samej figury i wektor  $\overrightarrow{w} \overrightarrow{prev\_w}$  jest skierowany do wnętrza wielokąta.

Jeśli zachodzi którakolwiek z opisanych powyżej sytuacji, to  $w$  jest niewidoczny. W przeciwnym wypadku punkt ten jest widoczny.



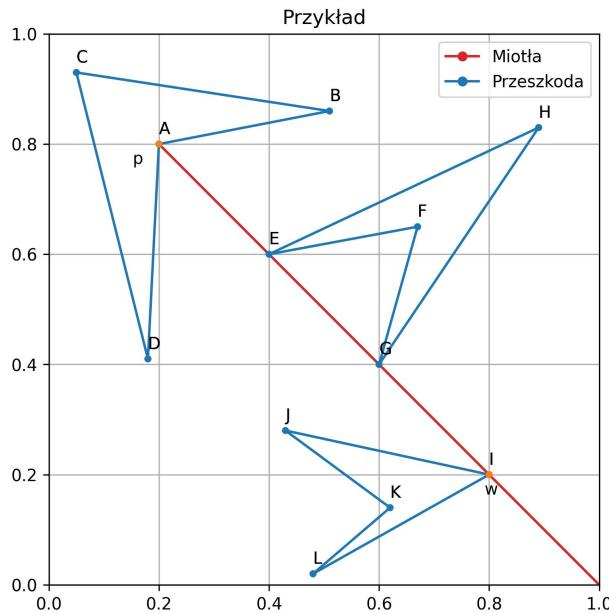
Rysunek 6: Przykład 1 dla przypadku 5

Punkt  $D$  był rozważany bezpośrednio przed punktem  $w$  i jest widoczny. Punkty  $p, D, w$  są wspólniowe. W strukturze stanu znajduje się odcinek  $IJ$ , który przecina  $Dw$ . Wynika z tego, że  $w$  jest niewidoczny.



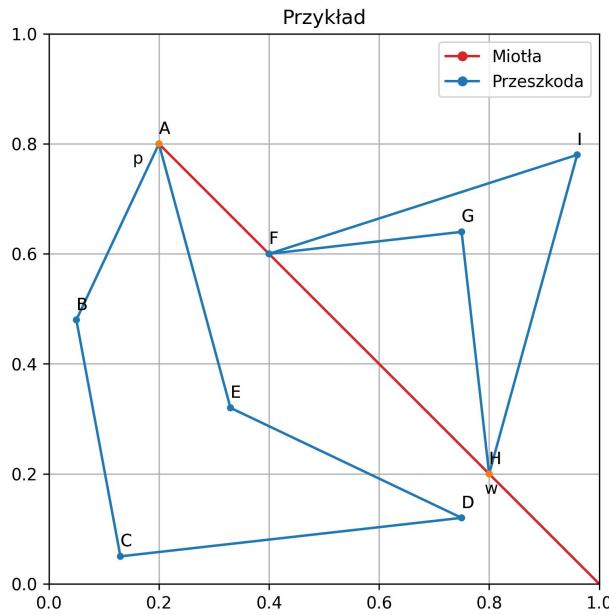
Rysunek 7: Przykład 2 dla przypadku 5

Punkt  $D$  był rozważany bezpośrednio przed  $w$  i jest widoczny,  $p, D, w$  są wspólniowe. W strukturze stanu nie znajduje się odcinek, który przecina  $Dw$ , ale  $w$  i  $D$  należą do tej samej figury i wektor  $\overrightarrow{wD}$  jest skierowany do wnętrza wielokąta, zatem  $w$  jest niewidoczny.



Rysunek 8: Przykład 3 dla przypadku 5

Punkt  $G$  był rozważany bezpośrednio przed punktem  $w$  i jest widoczny. Punkty  $p, G, w$  są współliniowe. W strukturze stanu nie znajduje się odcinek, który przecina  $Gw$ . Punkty  $w$  i  $G$  należą do różnych figur. Wynika z tego, że  $w$  jest widoczny.



Rysunek 9: Przykład 4 dla przypadku 5

Punkt  $F$  był rozważany bezpośrednio przed punktem  $w$  i jest widoczny. Punkty  $p, F, w$  są współliniowe. W strukturze stanu nie znajduje się odcinek, który przecina  $Fw$ . Punkty

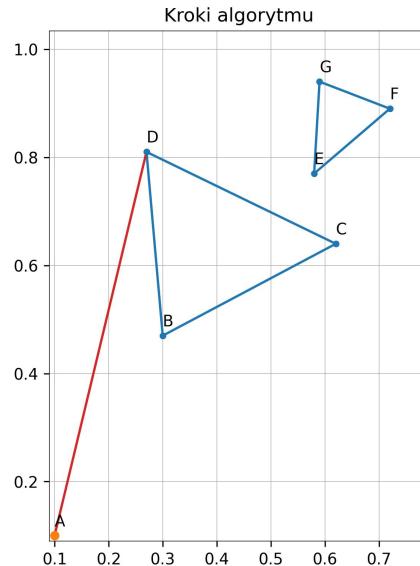
$w$  i  $F$  należą do tej samej figury, ale wektor  $\overrightarrow{wF}$  nie jest skierowany do wnętrza wielokąta, zatem  $w$  jest widoczny.

**Złożoność** Sprawdzenie, czy wektor pomiędzy dwoma wierzchołkami tej samej figury jest skierowany do środka wielokąta, dokonujemy przed wykonaniem algorytmu wyznaczania grafu widoczności. Nie pogarsza to złożoności całego algorytmu, ponieważ sprawdzenia takiego dokonujemy w złożoności  $O(n^2)$ . Dzięki temu podczas sprawdzania widoczności jesteśmy w stanie uzyskać informację czy dany wektor jest skierowany do wnętrza figury w czasie  $O(1)$ . Najbardziej kosztowna jest operacja znalezienia odcinka w strukturze stanu (Przypadek 3 i 5), wymaga ona czasu  $O(\log(n))$ . Wszystkie pozostałe operacje, takie jak sprawdzenie przecinania się odcinków, czy też sprawdzenie wspólniowości 3 punktów wymaga czasu  $O(1)$ .

Z powyższej analizy wynika, że czas potrzebny na stwierdzeniem, czy dany wierzchołek jest widoczny, to  $O(\log(n))$ .

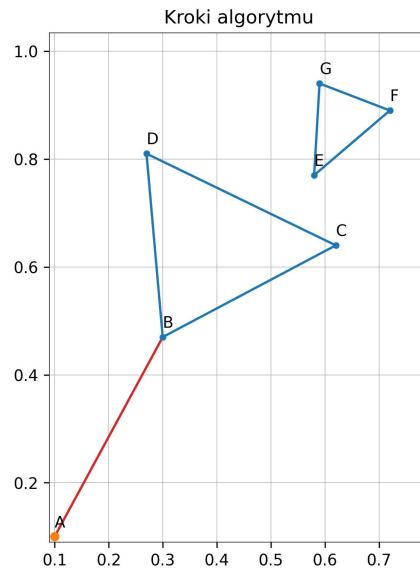
### 3.6 Kroki działania algorytmu

Punkt  $A$  jest punktem zaczepienia miotły. Miotła jest reprezentowana jako czerwony odcinek z  $A$  do obecnie przetwarzanego punktu.



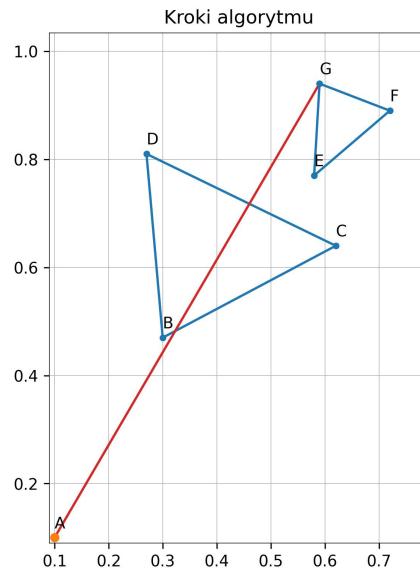
Rysunek 10: Przykład działania

Algorytm napotyka pierwsze zdarzenie. Punkt  $D$  jest widoczny, ponieważ struktura stanu jest pusta. Odcinki incydentne  $DB$  i  $DC$  dodawane są do struktury stanu (w tej kolejności).



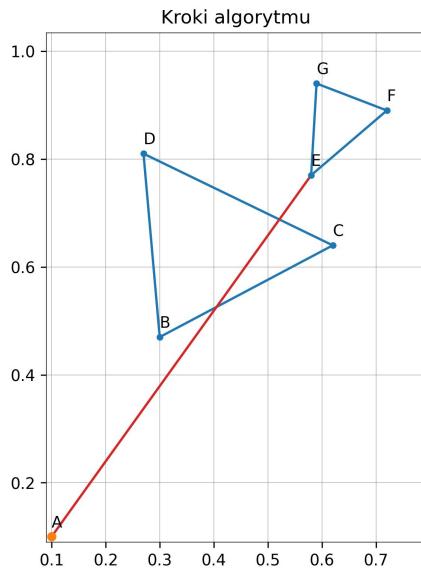
Rysunek 11: Przykład działania

$DB$  jest usuwane ze struktury,  $BC$  zostaje dodane. Punkt  $B$  jest widoczny, ponieważ nie istnieje krawędź przecinająca miotłę pomiędzy  $A$  i  $B$ .



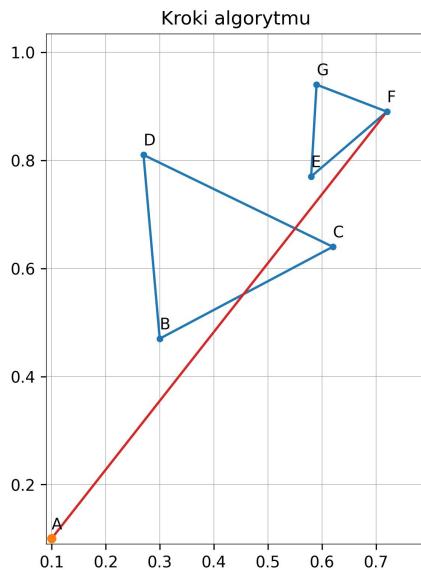
Rysunek 12: Przykład działania

Dodajemy do struktury  $GF$  i  $GE$ . Punkt  $G$  nie jest widoczny, bo zasłania go krawędź  $BC$  (pierwsza przecinająca miotłę).



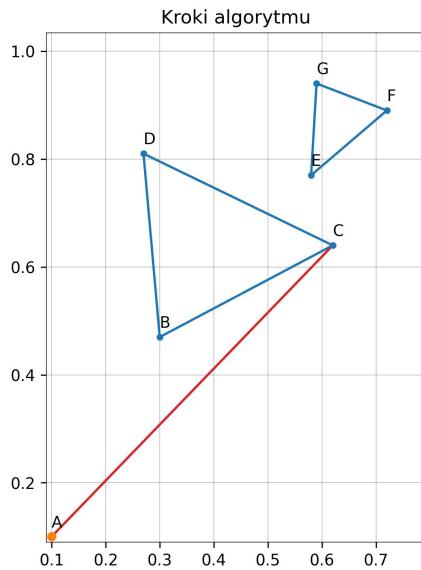
Rysunek 13: Przykład działania

Usuwamy  $GE$ , dodajemy  $EF$ . Analogicznie do  $G$ , punkt  $E$  nie jest widoczny.



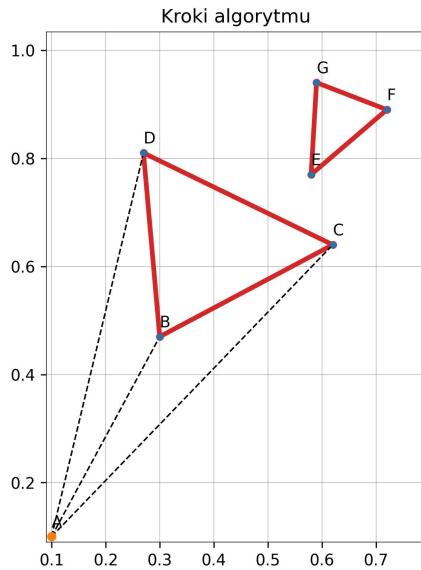
Rysunek 14: Przykład działania

Usuwamy  $EF$  i  $GF$ . Punkt  $F$  niewidoczny.



Rysunek 15: Przykład działania

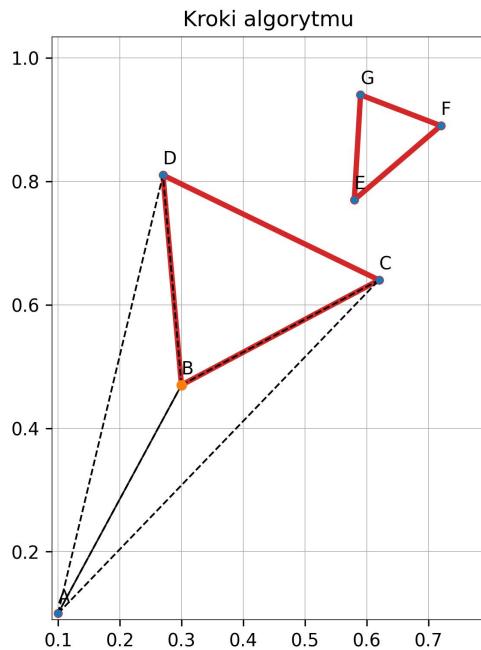
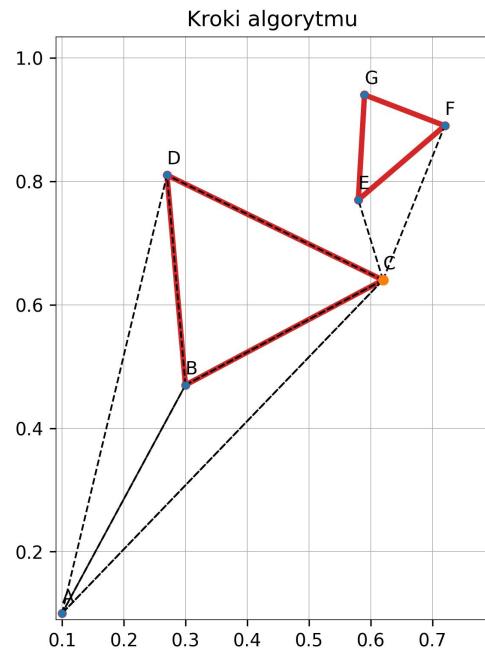
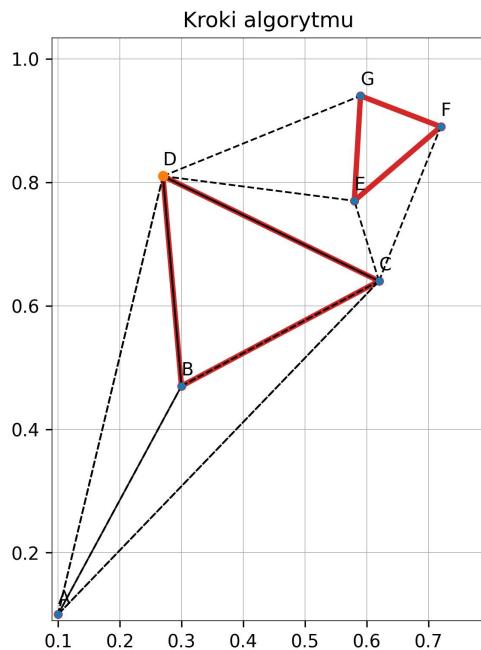
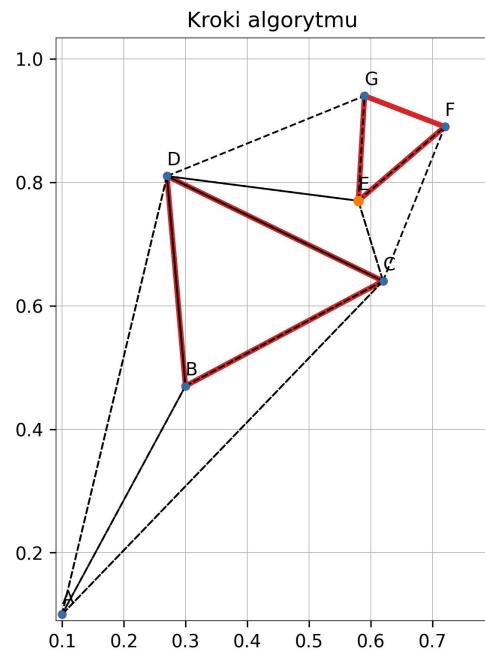
Usuwamy  $BC$  i  $DC$ . Punkt  $C$  widoczny.

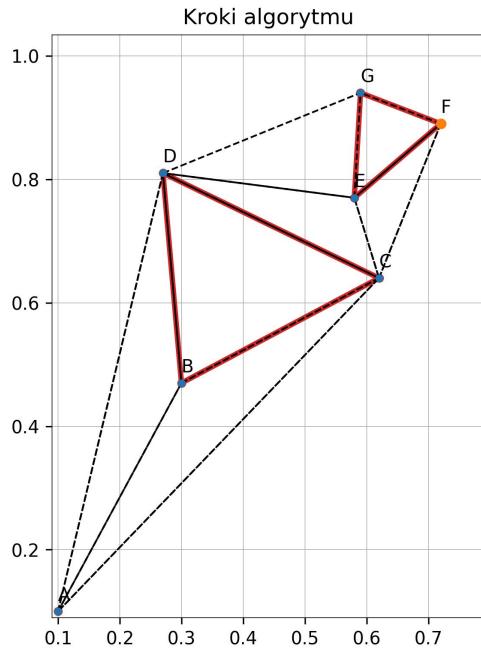
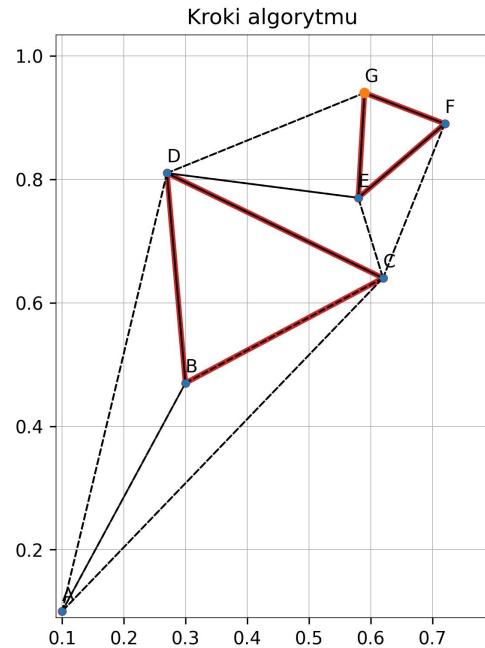


Rysunek 16: Wynik działania

Tak wygląda wynik wykonania powyższych kroków. Następnie wybierany jest nowy punkt zaczepienia i algorytm powtarza kroki wykonane powyżej.

Kolorem czerwonym oznaczone są figury (przeszkody), krawędzie dodane do grafu widoczności zaznaczone zostały przerywanymi liniami.

Rysunek 17: Miotła zaczepiona w  $B$ Rysunek 18: Miotła zaczepiona w  $C$ Rysunek 19: Miotła zaczepiona w  $D$ Rysunek 20: Miotła zaczepiona w  $E$

Rysunek 21: Miotła zaczepiona w  $F$ Rysunek 22: Miotła zaczepiona w  $G$ 

Tak wygląda ostatecznie graf widoczności dla tego przypadku.

### 3.7 Reprezentacja grafu

Graf jest zwracany przez funkcję `create_visibility_graph` jako obiekt klasy `VisibilityGraph`. Wewnątrz klasy graf jest reprezentowany jako słownik słowników. Dla każdego wierzchołka (reprezentowanego jako obiekt klasy `Point`) przechowywany jest słownik jego sąsiadów (widocznych punktów), gdzie kluczem jest sąsiadujący wierzchołek (również reprezentowany jako obiekt klasy `Point`), a wartością jest odległość tych dwóch punktów na płaszczyźnie. Graf w takiej postaci zwracany jest przez metodę `get_graph`.

Klasa posiada również metodę `get_lines`, która zwraca listę krawędzi grafu w postaci par punktów [*pierwszy punkt, drugi punkt*]. Punkty te są w tej liście reprezentowane jako dwuelementowe krotki postaci (współrzędna  $x$  punktu, współrzędna  $y$  punktu).

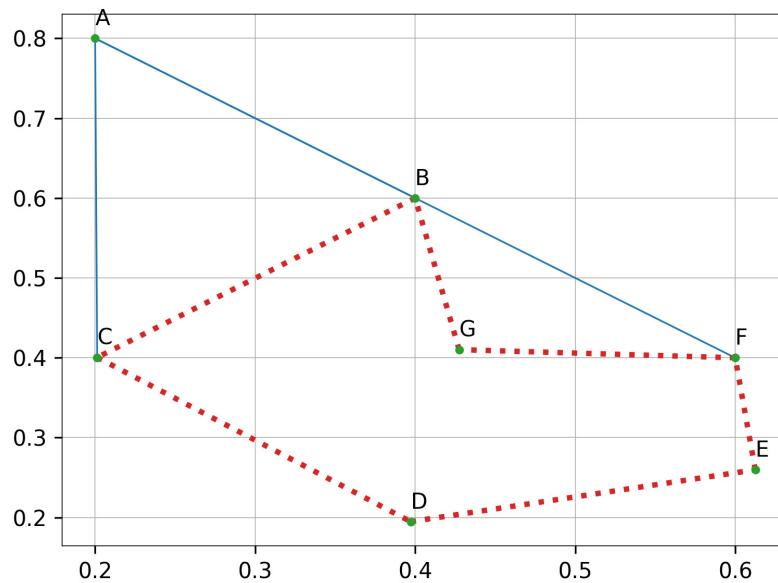
Poniżej w Tablicy 1 przedstawiony został graf widoczności dla przypadku opisywanego w punkcie 3.6, w postaci listy krawędzi.

Tablica 1: Reprezentacja grafu widoczności poprzez pary punktów tworzące listę krawędzi

$x_1$	$y_1$	$x_2$	$y_2$
0.1	0.1	0.27	0.81
0.1	0.1	0.3	0.47
0.1	0.1	0.62	0.64
0.3	0.47	0.1	0.1
0.3	0.47	0.27	0.81
0.3	0.47	0.62	0.64
0.62	0.64	0.1	0.1
0.62	0.64	0.3	0.47
0.62	0.64	0.27	0.81
0.62	0.64	0.58	0.77
0.62	0.64	0.72	0.89
0.27	0.81	0.58	0.77
0.27	0.81	0.62	0.64
0.27	0.81	0.3	0.47
0.27	0.81	0.1	0.1
0.27	0.81	0.59	0.94
0.58	0.77	0.62	0.64
0.58	0.77	0.27	0.81
0.58	0.77	0.59	0.94
0.58	0.77	0.72	0.89
0.72	0.89	0.62	0.64
0.72	0.89	0.58	0.77
0.72	0.89	0.59	0.94
0.59	0.94	0.72	0.89
0.59	0.94	0.58	0.77
0.59	0.94	0.27	0.81

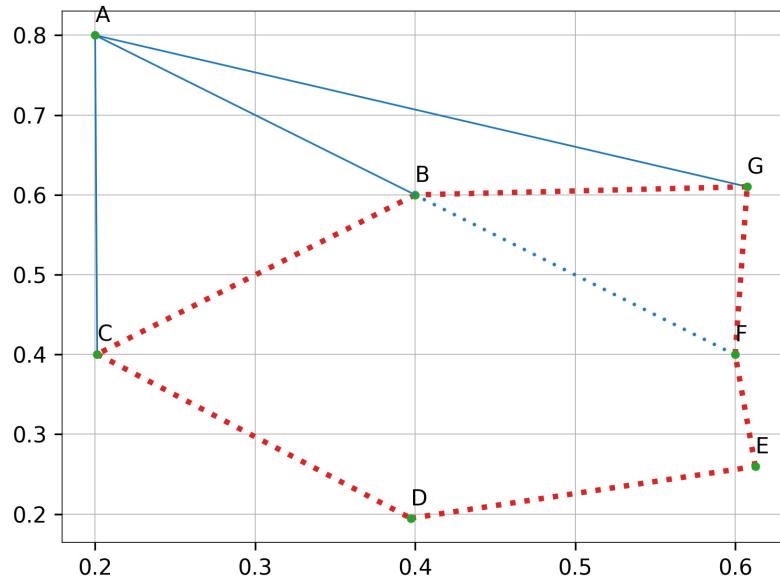
### 3.8 Testy

Na poniższych wykresach przedstawione zostały wyniki testów działania algorytmu, w postaci graficznej. Przedstawiają one grafy widoczności, gdzie wierzchołki zaznaczono kolorem zielonym, a krawędzie niebieskimi liniami. Figury stanowiące przeskody również zostały zawarte na wykresach i są oznaczone kolorem czerwonym. W testach 1-6 pokazane zostały krawędzie grafu widoczności tylko dla jednego wybranego wierzchołka oraz testy te zawierają dodatkowy opis przedstawianej sytuacji.



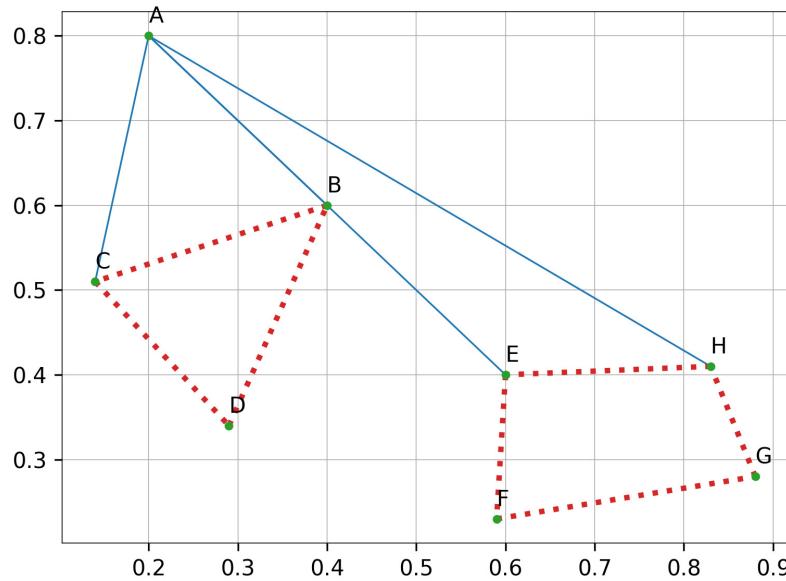
Rysunek 23: Przypadek testowy 1

Wierzchołek  $F$  jest widoczny, ponieważ  $B$  jest widoczny i miotły nie przecinają żadne krawędzie na odcinku  $BF$  oraz wektor  $\overrightarrow{FB}$  nie jest skierowany do wnętrza figury.



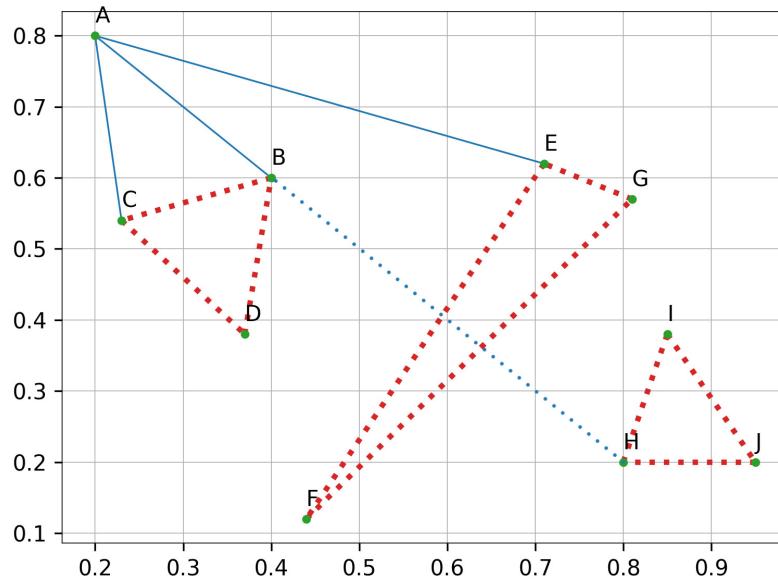
Rysunek 24: Przypadek testowy 2

Tutaj wektor  $FB$  jest skierowany do wnętrza figury więc punkt  $F$  jest niewidoczny (przerwana linia ma za zadanie tylko podkreślić fakt, że punkty  $B$  i  $F$  są współliniowe).



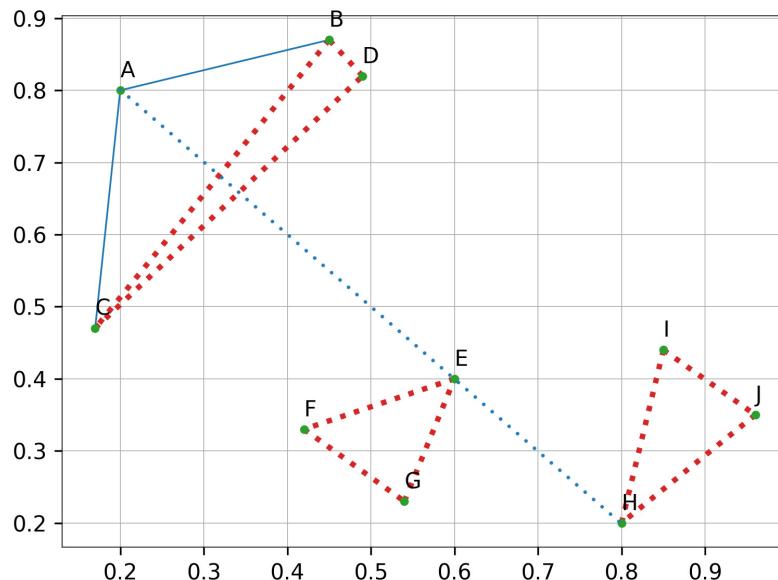
Rysunek 25: Przypadek testowy 3

Punkt  $E$  jest widoczny, ponieważ  $B$  jest widoczny i należą do różnych figur oraz pomiędzy nimi nie ma żadnej krawędzi przecinającej odcinek  $BE$ .



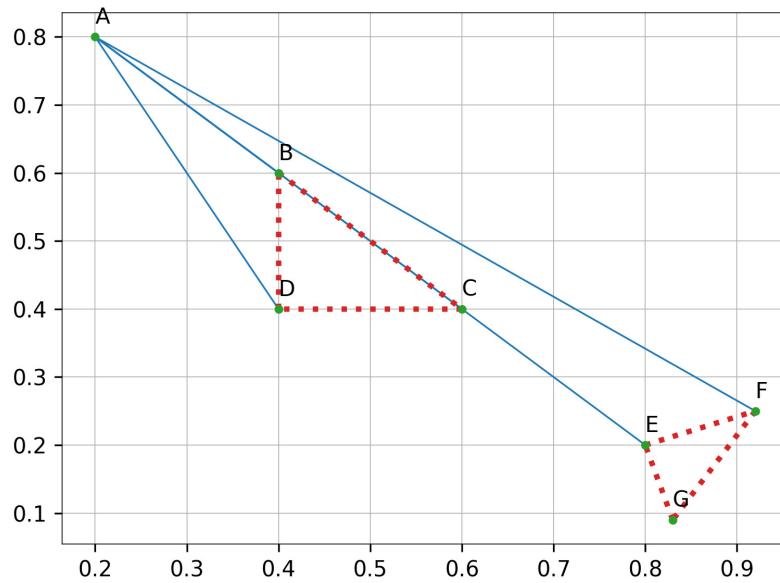
Rysunek 26: Przypadek testowy 4

Tutaj analogicznie jak wyżej,  $B$  jest widoczny i punkty  $B$  i  $H$  należą do różnych figur, ale  $BH$  jest przecięty dwoma odcinkami więc  $H$  nie będzie widoczny.



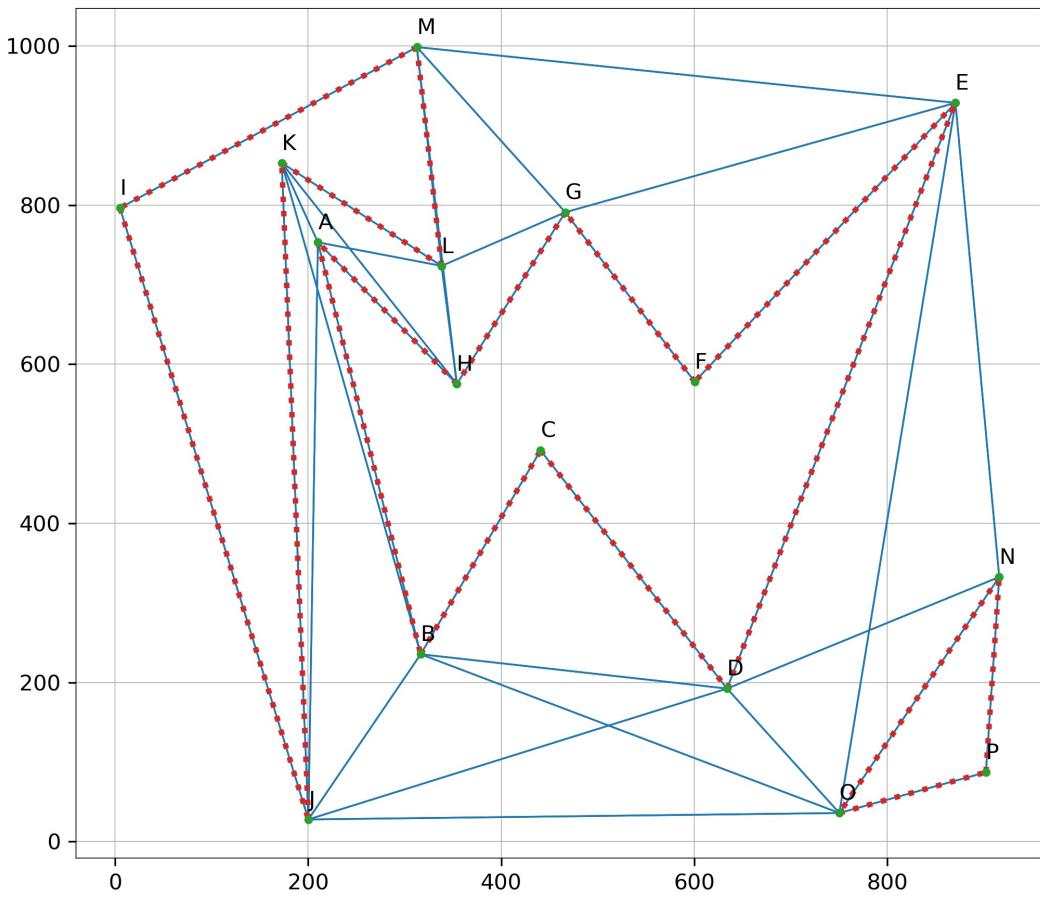
Rysunek 27: Przypadek testowy 5

Jeżeli  $E$  jest niewidoczny to nie ma sensu badać widoczności  $H$ , ponieważ jest on dalej od punktu zaczepienia miotły (punktu  $A$ ).

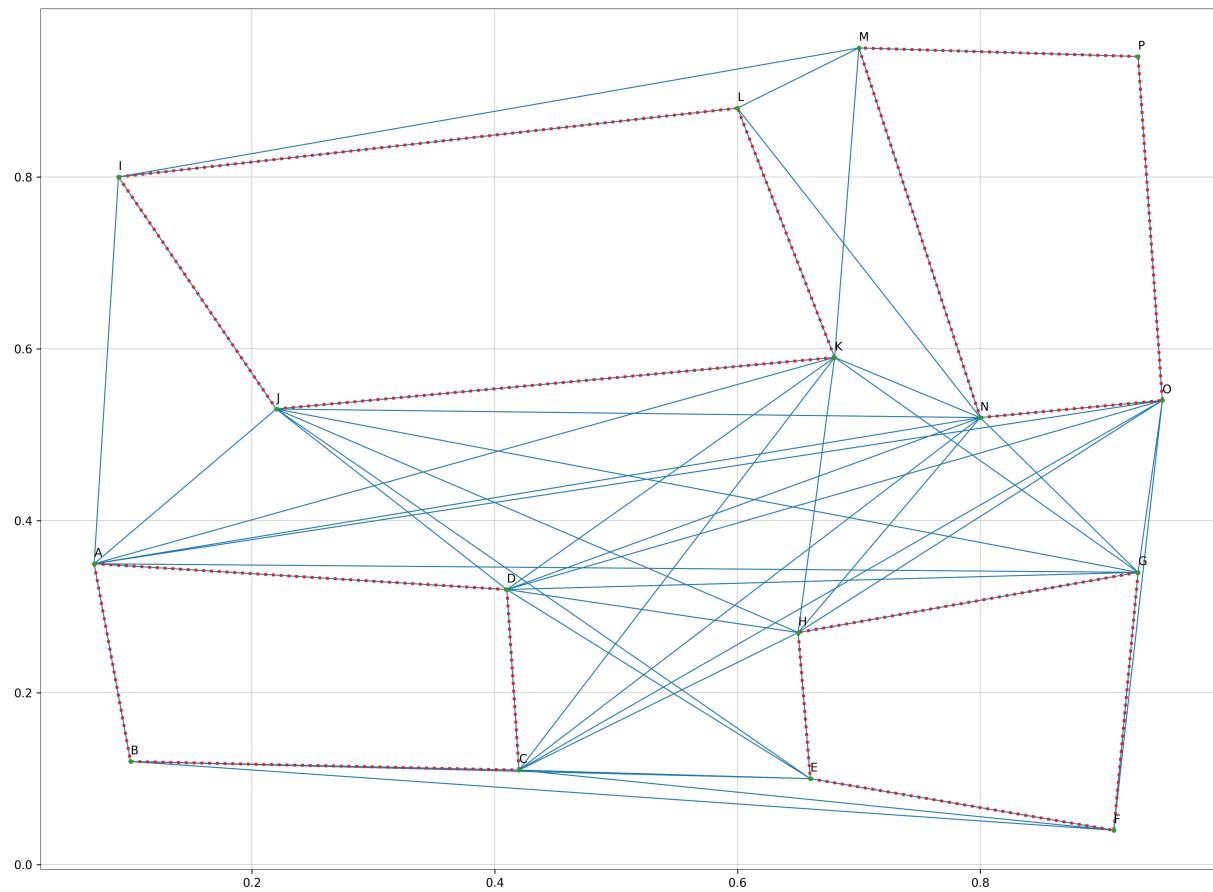


Rysunek 28: Przypadek testowy 6

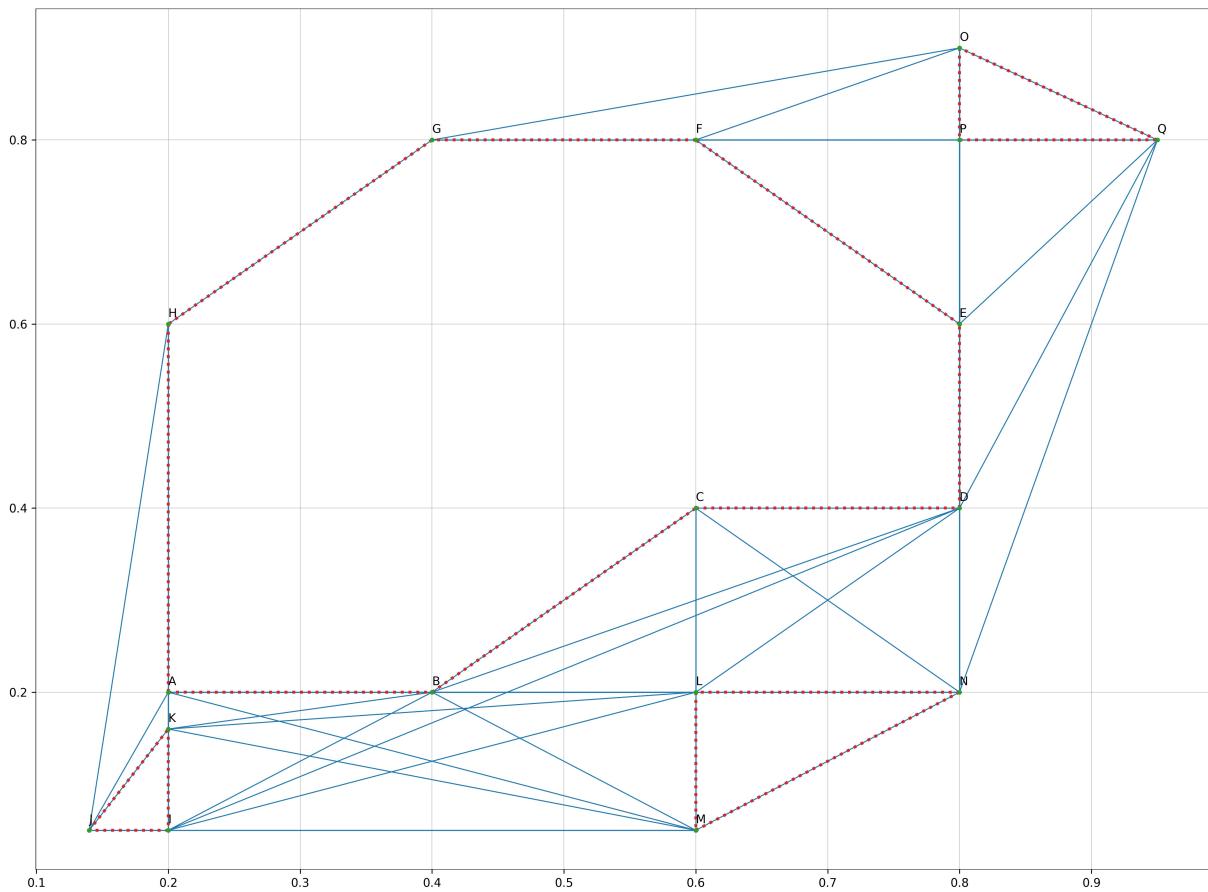
Wierzchołek  $E$  widoczny, gdy miotła pokrywa się z krawędzią  $BC$ .



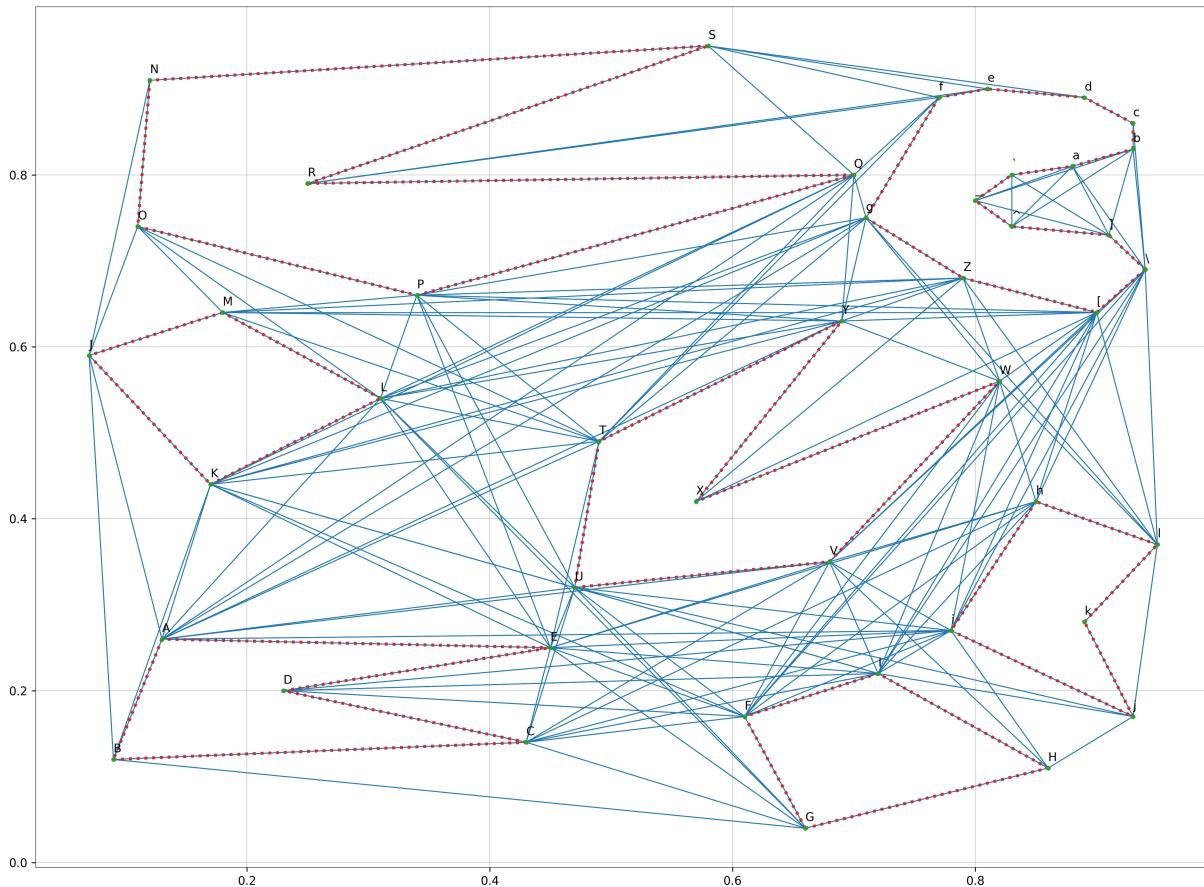
Rysunek 29: Przypadek testowy 'Przykład 1' zawarty w pliku Jupyter Notebook



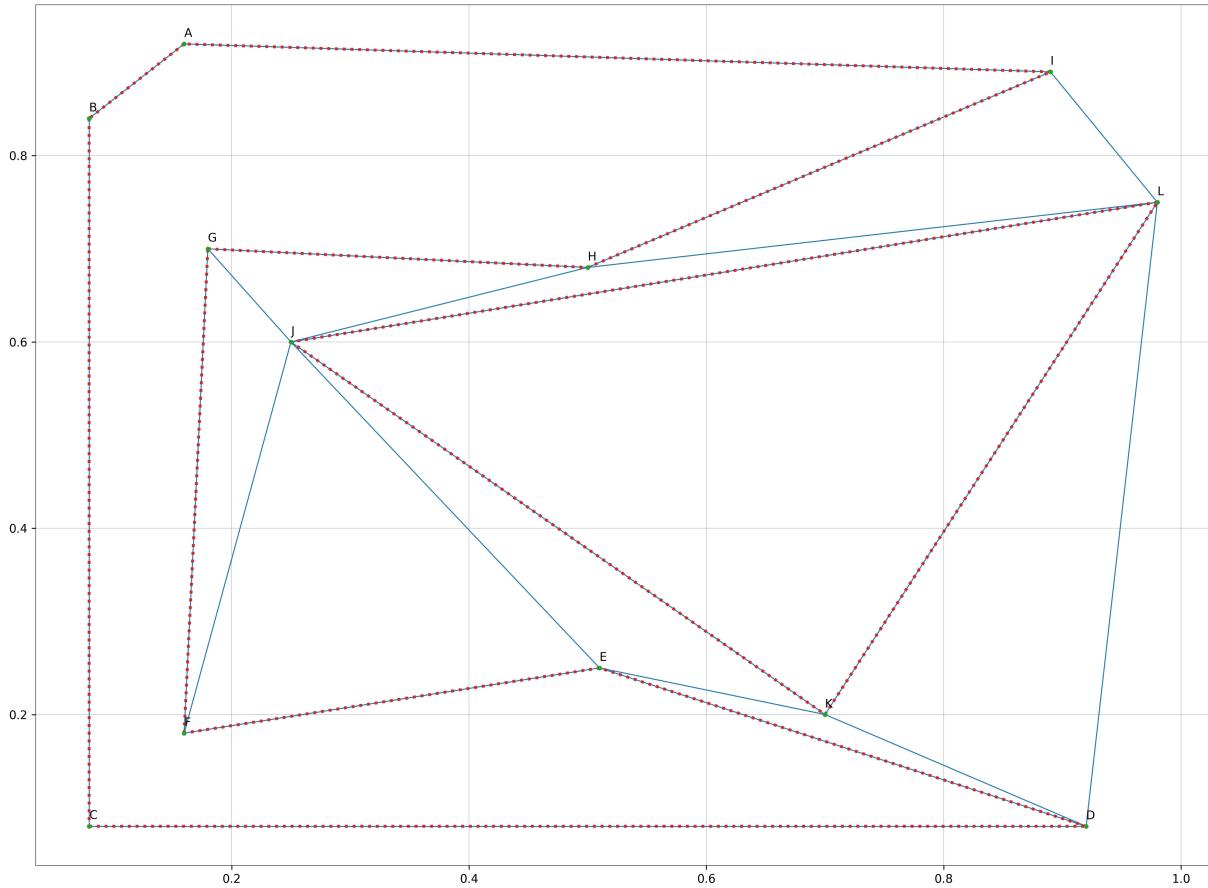
Rysunek 30: Przypadek testowy 7



Rysunek 31: Przypadek testowy 8



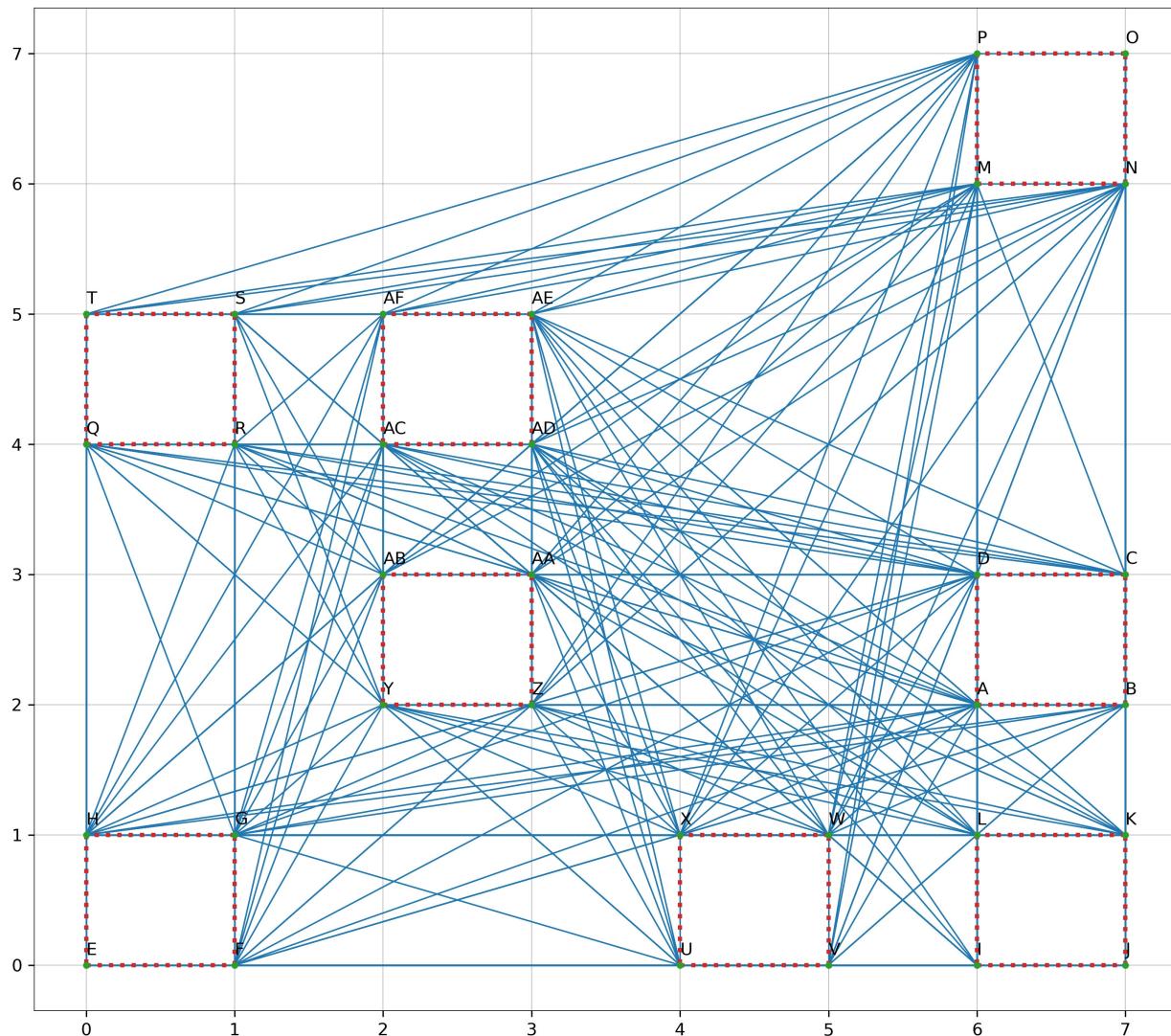
Rysunek 32: Przypadek testowy 9



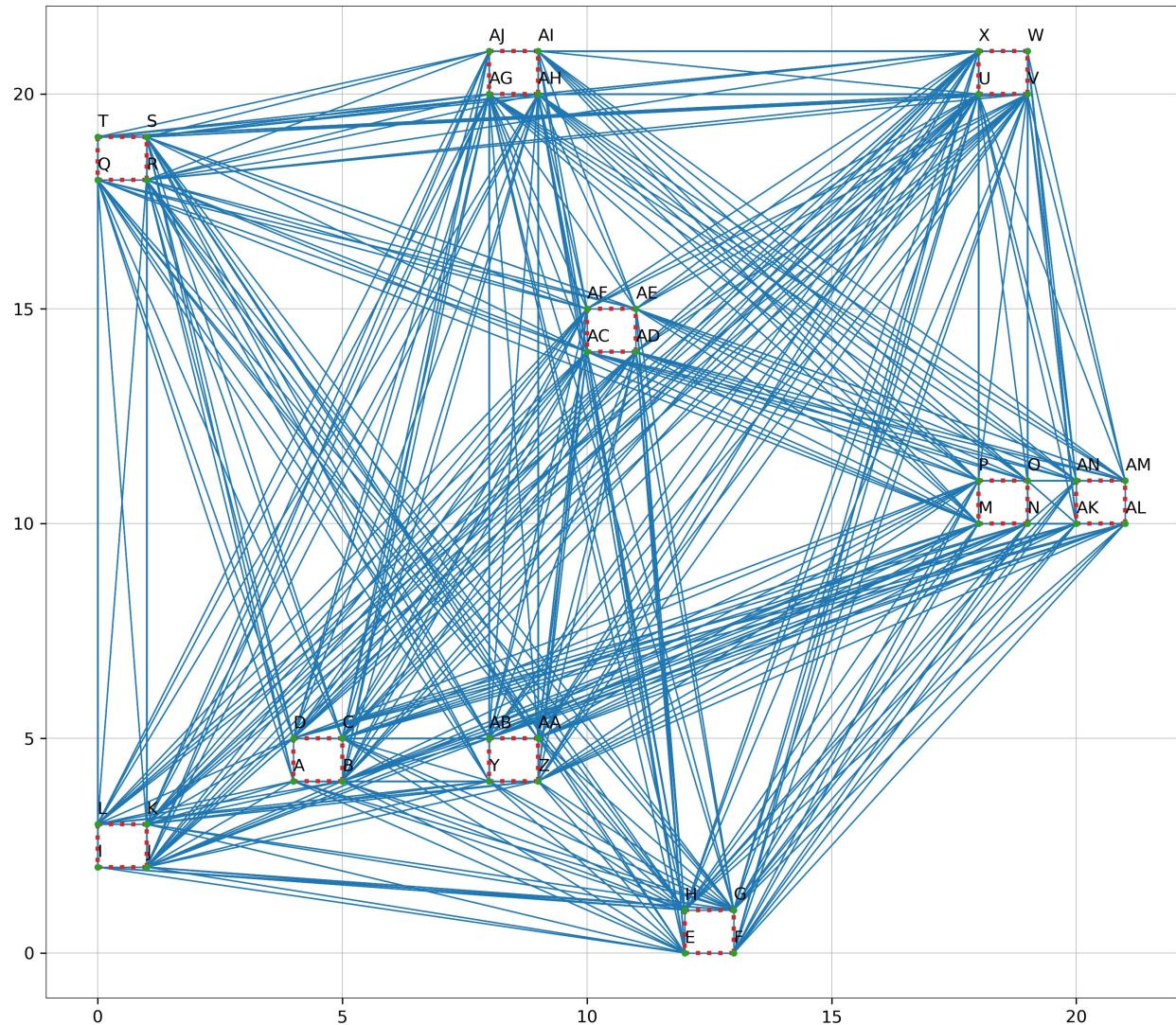
Rysunek 33: Przypadek testowy 10

### 3.9 Testy losowe

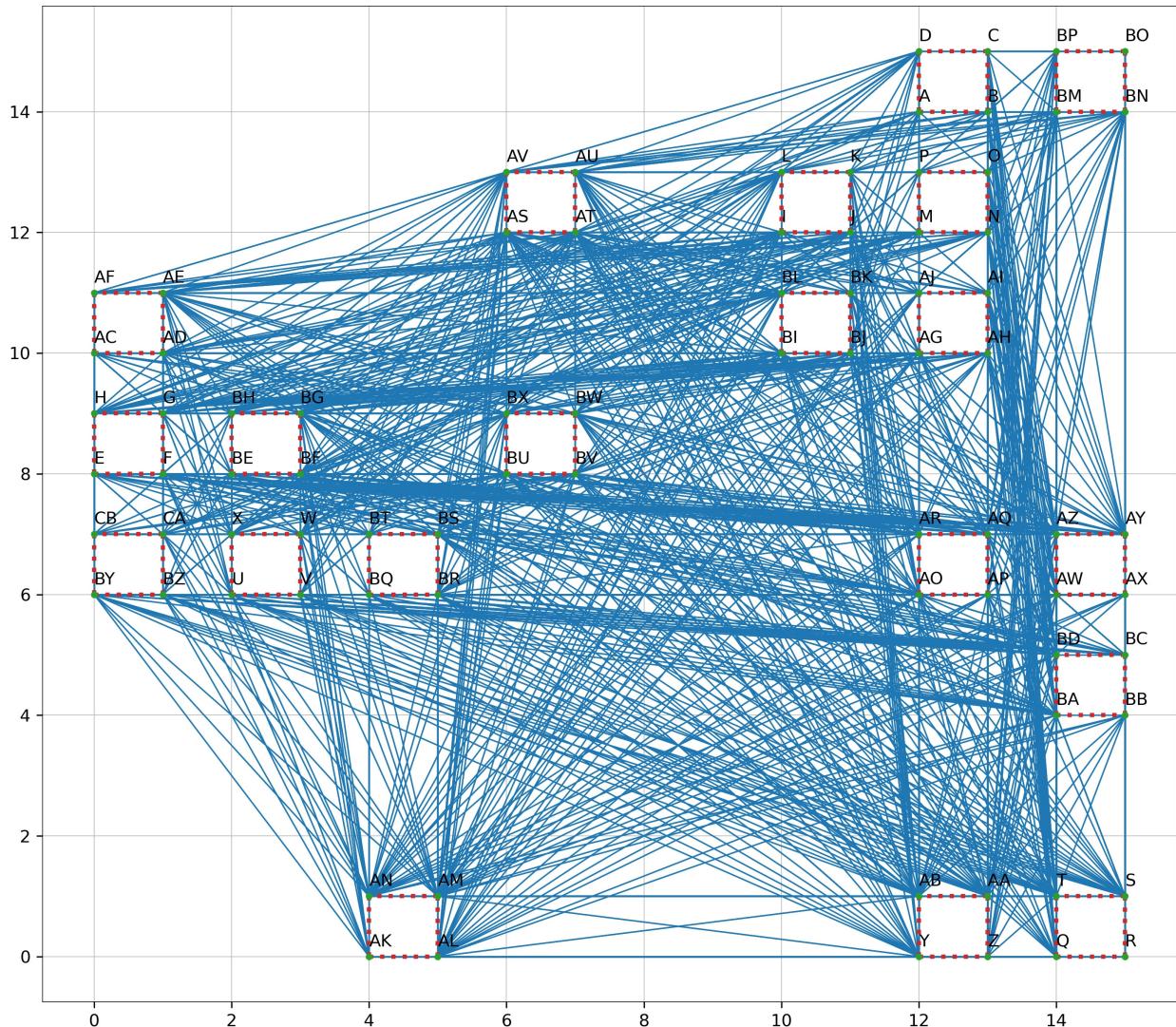
Poniższe ilustracje przedstawiają losowe testy wykonane przy użyciu modułu plot\_tool\_helpers.py. Testy polegają na generowaniu zadanej liczby kwadratów o wymiarach 1x1 i rozstawieniu ich w sposób losowy na płaszczyźnie o danych wymiarach.



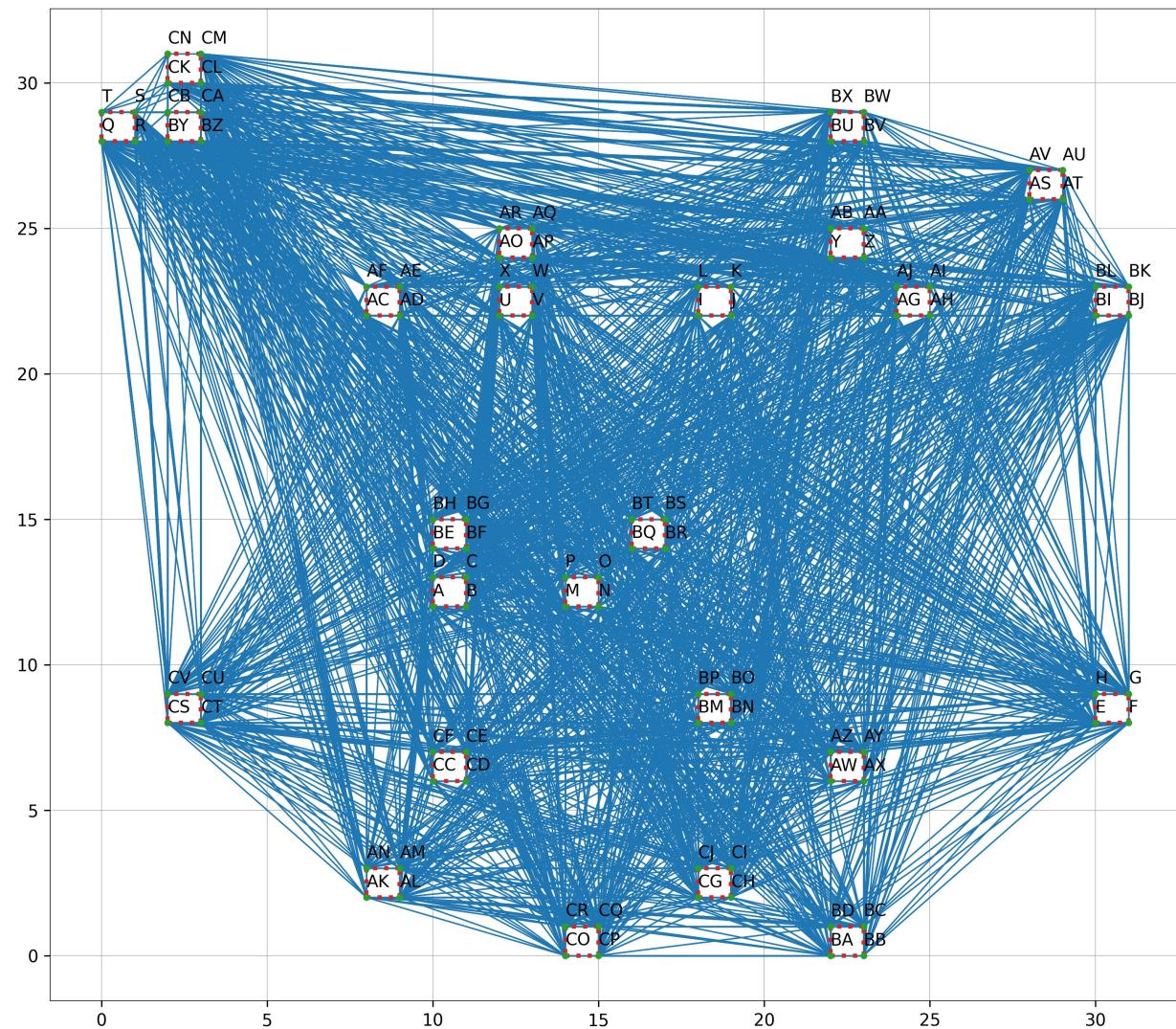
Rysunek 34: 8 figur na płaszczyźnie  $7 \times 7$



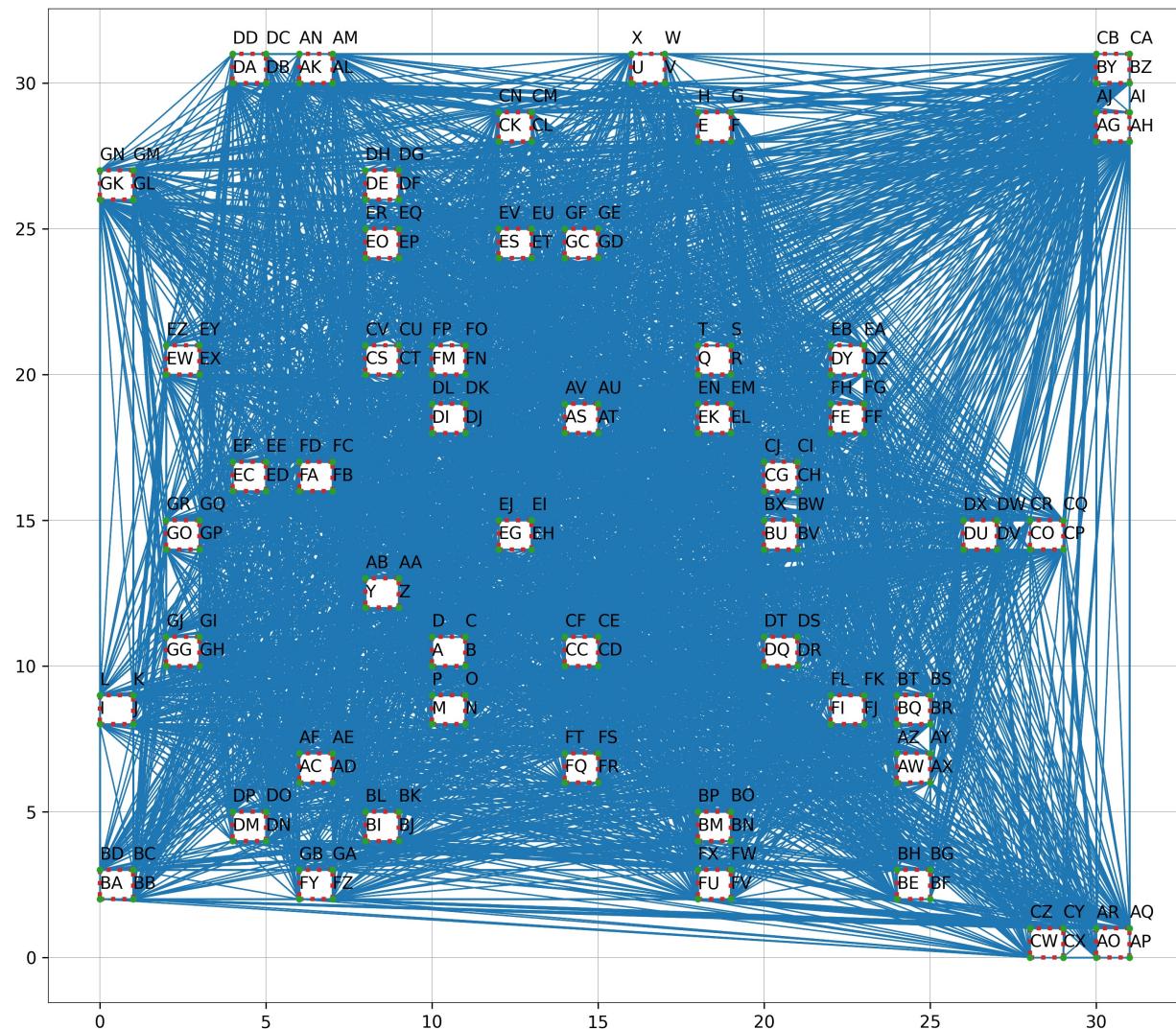
Rysunek 35: 10 figur na płaszczyźnie 21 x 21



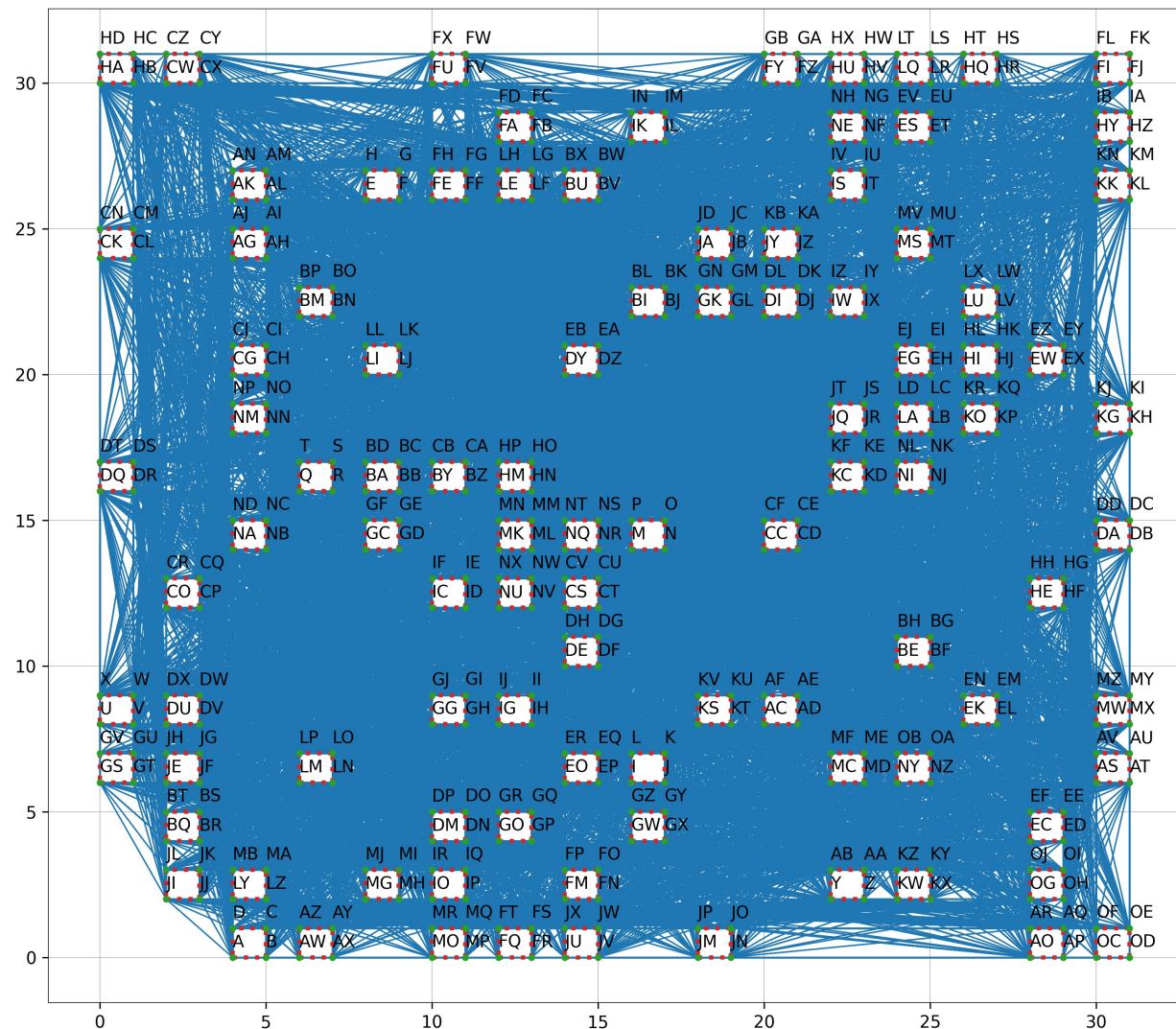
Rysunek 36: 20 figur na płaszczyźnie 16 x 16



Rysunek 37: 25 figur na płaszczyźnie 31 x 31



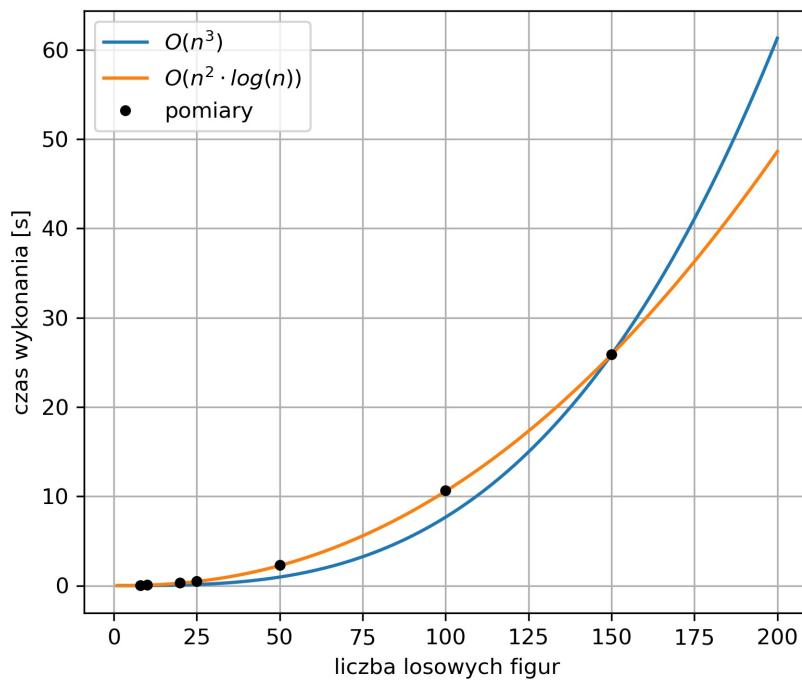
Rysunek 38: 50 figur na płaszczyźnie 31 x 31



Rysunek 39: 100 figur na płaszczyźnie 31 x 31

### 3.9.1 Czasy wykonania dla testów losowych

- 8 figur na płaszczyźnie 7 x 7 – 0.04488 sekund
- 10 figur na płaszczyźnie 21 x 21 – 0.07081 sekund
- 20 figur na płaszczyźnie 16 x 16 – 0.31419 sekund
- 25 figur na płaszczyźnie 31 x 31 – 0.48368 sekund
- 50 figur na płaszczyźnie 31 x 31 – 2.2919 sekund
- 100 figur na płaszczyźnie 31 x 31 – 10.62454 sekund
- 150 figur na płaszczyźnie 36 x 36 – 25.89075 sekund

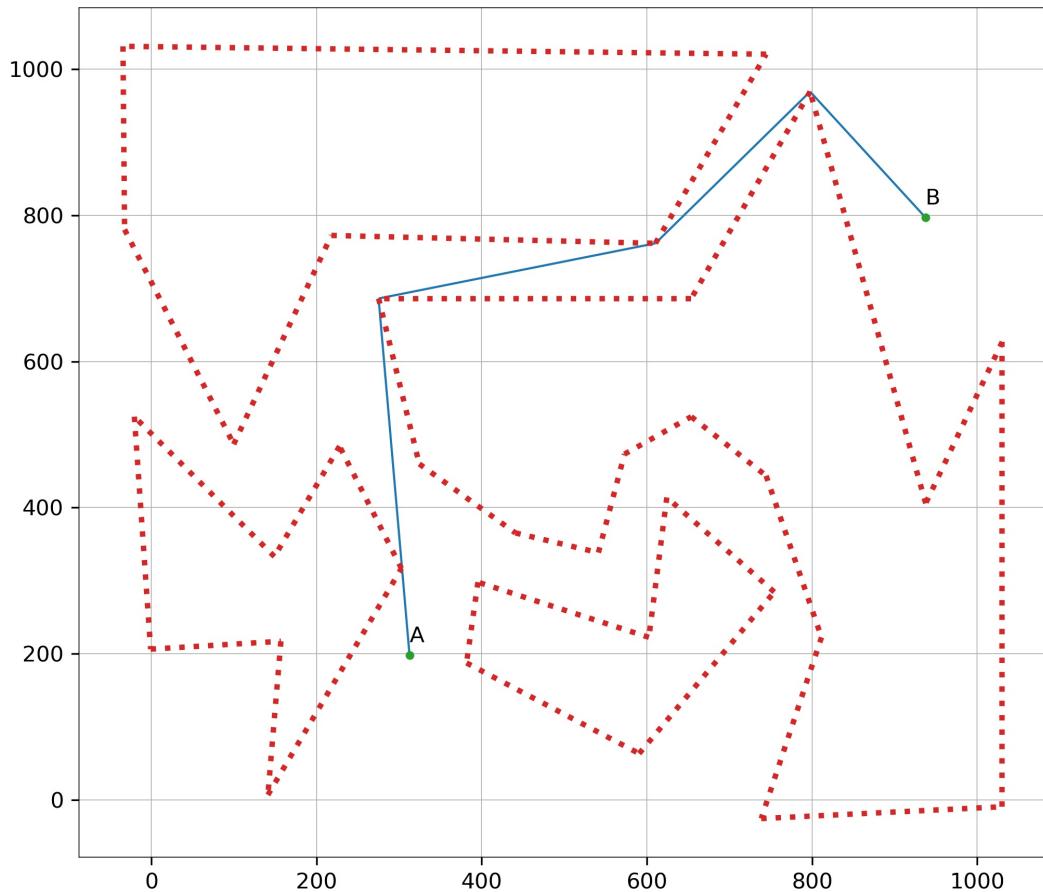


Rysunek 40: Czasy wykonania dla losowych figur

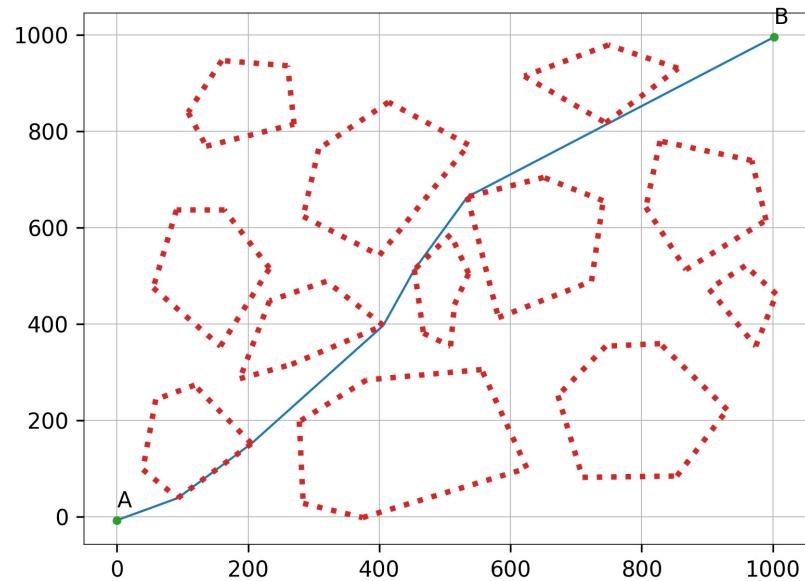
Generowanymi losowo figurami są kwadraty, zatem liczba wierzchołków  $n$  jest proporcjonalna do liczby figur. Możemy zatem dopasować krzywą, aby oszacować otrzymaną złożoność algorytmu. Jak widać na powyższym wykresie, po dopasowaniu krzywych do ostatniego pomiaru, otrzymane czasy niemal idealnie pokrywają się z krzywą  $n^2 \log(n)$ , co świadczy o tym, że złożoność algorytmu jest zgodna z oczekiwana.

### 3.10 Testy znajdowania najkrótszej ścieżki

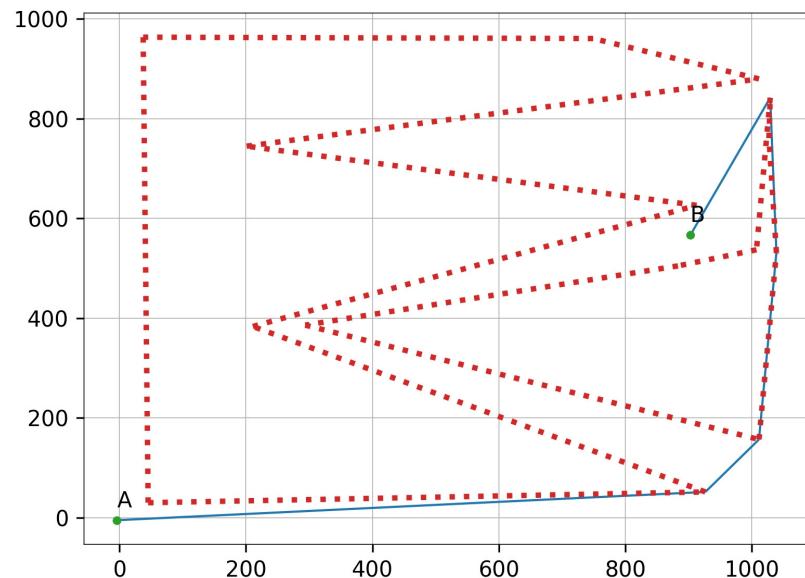
Poniżej przedstawiono przykłady zastosowania grafu widoczności do wyznaczenia najkrótszej ścieżki pomiędzy dwoma punktami na płaszczyźnie, na której znajdują się przeszkody w postaci wielokątów. Niebieską linią zaznaczono wyznaczoną ścieżkę, natomiast przeszkody oznaczono czerwoną przerywaną linią. Do szukania najkrótszej ścieżki w grafie zastosowane algorytm Dijkstry.



Rysunek 41: Przypadek testowy 'Przykład 2' zawarty w pliku Jupyter Notebook



Rysunek 42: Przypadek testowy



Rysunek 43: Przypadek testowy

## 4 Wnioski

- We wszystkich testowanych przypadkach, algorytm poprawnie wyznaczył graf widoczności. Wyniki testów w postaci graficznej zostały przedstawione w punkcie 3.8.
- Dzięki zastosowaniu algorytmu zamiatania i struktury zrównoważonego drzewa binarnego, udało się obniżyć złożoność z  $O(n^3)$  dla algorytmu trywialnego, do złożoności  $O(n^2 \cdot \log(n))$ , czego potwierdzeniem są wyniki testów czasowych w punkcie 3.9.1.
- Przy użyciu wyznaczonego grafu widoczności udało się zaimplementować algorytm wyznaczania najkrótszej ścieżki pomiędzy dwoma punktami na płaszczyźnie, na której znajdują się przeszkody w postaci wielokątów. Kilka przykładów jego działania przedstawiono w punkcie 3.10.

## 5 Bibliografia

Algorytm został przygotowany na podstawie książki Marka de Berga pt. *Geometria Obliczeniowa - Algorytmy i Zastosowania*.