Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Anna Sarnavska

Nr albumu: 1142373

Nakładanie map w geometrii obliczeniowej

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Informatyki Stosowanej

Kraków 2020

1:8907278395

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia   Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia   Podpis kierującego pracą

Chcę podziękować Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za pomoc, cierpliwość i wyrozumiałość podczas całego okresu pisania pracy, a także za cenne rady i ważne informacje, dzięki którym nauczyłam się wiele nowych, ciekawych rzeczy, które na pewno będą przydatne w przyszłości.

3:4846497472

Streszczenie

W pracy został opracowany i zaimplementowany w języku Python, algorytm z geometrii obliczeniowej, polegający na nakładaniu map. Algorytm przedstawia spis kroków, wymaganych do połączenia dwóch wejściowych map. Mapa jest to graf spójny płaski. Do reprezentacji abstrakcyjnego grafu została dodana struktura podwójnie powiązanych list krawędzi. W ten sposób jest możliwe obieganie każdej ściany mapy po podaniu dowolnej krawędzi do niej należącej.

Algorytm zawiera sprawdzanie wszystkich możliwości przecinania się dwóch krawędzi oraz przekształcenie kopii pierwszej wejściowej mapy w mapę, reprezentującą wynik nakładania.

Dodatkowo, przy opracowaniu algorytmu nakładania map, powstała metoda generująca krawędzie w taki sposób, aby mapa po dodaniu każdej kolejnej krawędzi miała postać spójną. Ten algorytm jest oparty na algorytmie przeszukiwania grafu wszerz. Kolejne nowo dodane metody to dodawanie liścia i cięciwy do mapy.

W implementacji wykorzystano następujące struktury geometryczne: punkt, odcinek, prostokąt, graf.

Słowa kluczowe: graf płaski, nakładanie map, geometria obliczeniowa, graf spójny, podwójnie wiązana lista krawędzi

4:1025296095

English title: Map overlay in computational geometry

Abstract

A computational geometry algorithm for map overlay has been developed and implemented in the thesis using Python. The algorithm represents steps, which are needed for overlaying two maps. A map is a plane connected graph. Doubly connected edge lists have been added to the representation of abstract graphs. This way it is possible to walk along the border of any face if one of its edges is given.

The algorithm checks all possible cases of the intersection of two edges and transforms a copy of the first input map into a maps overlaying result.

In addition, during the implementation of the map overlay algorithm, a new method has been added. This method generates edges sequentially and maintains map connectivity in each iteration. This algorithm is based on breadth-first search algorithm. Another newly added methods are adding a leaf and adding a chord to the map.

The following geometrical structures were used for implementation: point, segment, rectangle, graph.

Keywords: plane graph, map overlay, computational geometry, connected graph, doubly connected edge list

5:9382018928

Spis treści

Spis rysunków 3
Listings 4
1. Wstęp 5
1.1. Cele pracy 5
1.2. Organizacja pracy 6
2. Geometria obliczeniowa 7
3. Implementacja 9
3.1. Nowe elementy interfejsu grafów 11
3.2. Nowe elementy obiektów geometrycznych 12
4. Algorytmy 13
4.1. Algorytm sprawdzający przecinanie się map 13
4.2. Algorytm generowania krawędzi mapy w sposób spójny 14
4.3. Algorytm dodawania cięciwy do mapy 15
4.4. Algorytm dodawania liścia do mapy 16
4.5. Algorytm nakładania map 17
5. Podsumowanie 21
A. Kod źródłowy 22
B. Testy algorytmów 24
B.1. Test map bez przecinających się krawędzi. 24
B.2. Test map z równoległymi krawędziami 25
B.3. Test map z przecinającymi się krawędziami 25
B.4. Test złożoności wykonania algorytmu nakładania map 25
Bibliografia 37

6:4467750122

Spis rysunków

B.1. Test 1. Mapy wejściowe bez przecięcia. 24
B.2. Test2. Mapy wejściowe ze wspólną krawędzią. 26
B.3. Test2. Mapa wyjściowa. 26
B.4. Test3. Mapy wejściowe. 27
B.5. Test3.Mapa wyjściowa. 27
B.6. Test4. Mapy wejściowe. 28
B.7. Test4. Mapa wyjściowa. 28
B.8. Test5. Mapy wejściowe. 29
B.9. Test5. Mapa wyjściowa. 29
B.10. Test6. Mapy wejściowe. 30
B.11. Test6. Mapa wyjściowa. 30
B.12. Test7. Mapy wejściowe. 31
B.13. Test7. Mapa wyjściowa. 31
B.14. Test8. Mapy wejściowe. 32
B.15. Test8. Mapa wyjściowa. 32
B.16. Test9. Mapy wejściowe z parametrem $S = 1$. 33
B.17. Test9. Mapa wyjściowa z parametrem $S = 1$. 33
B.18. Test9. Mapy wejściowe z parametrem $S = 5$. 34
B.19. Test9. Mapa wyjściowa z parametrem $S = 5$. 34
B.20. Test9. Mapy wejściowe z parametrem S = 10. 35
B.21. Test9. Mapa wyjściowa z parametrem S = 10. 35
B.22. Test9. Wykres wydajności nakładania map. 36

7:2725191431

Listings

3.1 Sesja interaktywna z odcinkami prostopadłymi. 9
4.1 Metoda Graph.__edge_of_intersection(). 13
4.2 Metoda Graph.iteredges_connected(). 14
4.3 Metoda Graph.add_chord(). 16
4.4 Metoda Graph.add_leaf(). 17
A.1 Metoda Graph.map_overlay(). 22
B.1 Test dla map bez przecięcia. 24

8:2349241877

1. Wstęp

Tematem niniejszej pracy jest algorytm nakładania map. Algorytm należy do geometrii obliczeniowej (ang. *computational geometry*) [1]. Jest to dziedzina, która zajmuje się działaniem na obiektach geometrycznych (w tym punktach, odcinkach, okręgach i wielokątach) przy użyciu komputerów, z wykorzystaniem odpowiednich struktur danych i algorytmów. W opracowanym algorytmie mapę reprezentuje się za pomocą grafu (ang. *graph*) [2], który jest płaski i spójny. Struktura mapy służy do reprezentacji obiektów geometrycznych i badania operacji na tych obiektach. Jedną z takich możliwych operacji jest łączenie dwóch map (nakładanie).

Algorytm w przyszłości można udoskonalić, tak aby służył w praktycznych zastosowaniach. Lista podwójnie wiązana nadaje możliwość przechowywania wielu cennych informacji: wierzchołki mogą odpowiadać obiektom takim jak miasta czy skrzyżowania, krawędzie mogą reprezentować ulice, linie kolejowe, rzeki, a ściany mogą reprezentować miasta, państwa, lasy, jeziora, morza.

W niniejszej pracy nakładamy na siebie mapy, które mają identyczną strukturę oraz nie niosą dodatkowej informacji o obiekcie, który one reprezentują. Można powiedzieć, że obecnie jest opracowany tylko szkielet nakładania map. Natomiast przy wykorzystaniu praktycznym, opisanym powyżej, warto rozważyć możliwości przechowywania informacji o tym, do której mapy należał wierzchołek, krawędź albo ściana przed nakładaniem. Przykład: pierwsza mapa reprezentuje rzeki, a druga reprezentuje drogi. Po nałożeniu na siebie tych map, dostajemy informację o przecięciu rzeki z drogą, co powinno odpowiadać mostowi na rzece. Most będzie wierzchołkiem nowej mapy, gdzie spotykają się krawędzie dwóch rodzajów (rzeka i droga).

1.1. Cele pracy

Głównym celem pracy było stworzenie algorytmu pozwalającego nakładać dwie mapy. W wyniku nałożenia dwóch map powstaje nowa trzecia mapa, która zawiera wierzchołki starych map oraz nowe wierzchołki powstałe na przecięciu krawędzi dwóch map. Nowa mapa może zawierać całe krawędzie starych map lub nowe krawędzie będące podzielonymi częściami pewnych krawędzi ze starych map. Każda kolejna operacja algorytmu powinna dodawać krawędzie w sposób, aby po każdej iteracji wynikowa mapa była spójna [3].

Warto dodać, że w pewne biblioteki zawierają operację nakładania map, ale nie znaleźliśmy udostępnionych kodów źródłowych takiego algorytmu. Opis w literaturze [4] był bardzo zwięzły, bez szczegółów implementacyjnych. Ponadto opierał się na zaawansowanej technice zamiatania płaszczyzny i roz-

9:9813963751

ważał ogólniejszy przypadek ścian z dziurami. Złożoność obliczeniowa tego algorytmu wynosi $O(n \log n + k \log n)$, gdzie n jest sumą złożoności map, a k jest złożonością nałożenia.

1.2. Organizacja pracy

Organizacja niniejszej pracy jest następująca. Rozdział 1 opisuje główne cechy algorytmu nakładania map oraz cele i organizację pracy. Rozdział 2 zawiera wprowadzenie do geometrii obliczeniowej, a także opis podstawowych definicji i twierdzeń. W rozdziale 3 zostały opisane założenia implementacyjne, nowe elementy interfejsu grafów oraz obiektów geometrycznych. Rozdział 4 omówiono implementację niezbędnych algorytmów oraz problem, który rozwiązuje algorytm nakładania map. Dodatek A przechowuje kod algorytmu. W dodatku B są przedstawione wyniki testów dla zaimplementowanych algorytmów.

10:8396809596

2. Geometria obliczeniowa

Podstawowe definicje i twierdzenia.

Opis grafu płaskiego: Podwójnie połączona lista krawędzi (ang. *doubly connected edge list*) [8] jest to struktura danych do reprezentowania grafu płaskiego, czyli grafu planarnego zanurzonego w płaszczyźnie. Dla prostoty rozważa się tylko grafy spójne, choć istnieją uogólnienia do opisu grafów niespójnych, w których ściany mają "dziury". Jeżeli przedstawiony na płaszczyźnie graf nie posiada krawędzi przecinających siebie lub wierzchołki, to jest to graf zanurzony.

Opis grafu spójnego: Graf spójny (ang. *connected*) [3] jest zbudowany tak, że między dowolnymi dwoma różnymi wierzchołkami istnieje ścieżka. W takim przypadku każdy wierzchołek, bezpośrednio lub pośrednio, jest połączony ze wszystkimi innymi.

Opis ściany grafu płaskiego: W grafie płaskim oprócz pojęć wierzchołka i krawędzi występuję również pojęcie ściany grafu (ang. *face*) [5]. Ścianą nazywamy zamknięty obszar płaszczyzny, którą ograniczają pewne krawędzie tego grafu. Ścianą można nazwać też nieograniczoną część płaszczyzny, która znajduje się na zewnątrz grafu - jest to ściana zewnętrzna. Każdy graf płaski posiada jedną ścianę zewnętrzną oraz skończoną liczbę ścian wewnętrznych (ograniczonych).

Problem przecinania się map: Dla poprawnego działania algorytmu nakładania map musimy sprawdzić, czy dwie mapy przecinają się. Należy znaleźć pewną krawędź jednej mapy, która przecina się z pewną krawędzią drugiej mapy. Przecinanie się odcinków można wykryć analizując wzajemne położenia końców odcinków, co zostało wykorzystane w metodzie `Segment.intersect()` (metoda była dostępna w bibliotece). Algorytm sprawdzania przecięcia map został opisany w sekcji 4.1.

Definicja liścia w mapie: Liśćmi (ang. *leaf node*) [11] grafu są nazywane wierzchołki stopnia jeden. Liście są także nazywane wierzchołkami zewnętrznymi. Dodawanie liścia do mapy oznacza dodawanie do mapy nowej krawędzi, której jeden koniec należy już do mapy, a drugi koniec jest nowym wierzchołkiem w mapie. Algorytm dodawania liścia do mapy jest opisany w rozdziale 4.4.

Definicja cięciwy w mapie: Cięciwa w mapie jest to nowa krawędź w mapie, która łączy dwa istniejące wierzchołki mapy. Algorytm dodawania cięciwy do mapy jest opisany w rozdziale 4.3.

11:5192798800

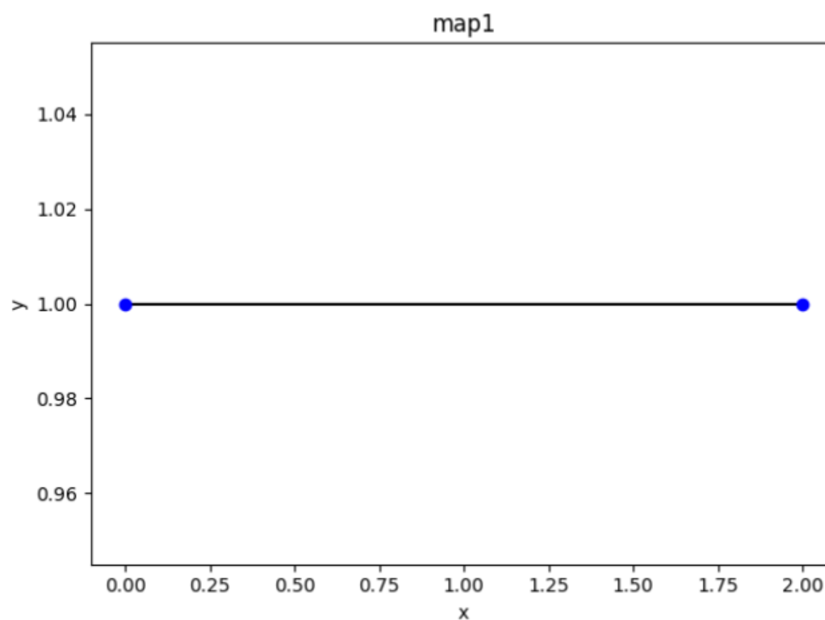
Algorytm BFS: Przeszukiwanie grafu wszerz (ang. *breadth first search*) [10] jest to algorytm pozwalający przejść przez graf. Zaczyna się on od zadanego wierzchołka początkowego, po czym następuje odwiedzanie wierzchołków osiągalnych z niego. Algorytm korzysta z kolejki FIFO, a jego złożoność obliczeniowa wynosi $O(|V| + |E|)$, gdzie $|V|$ to liczba wierzchołków, $|E|$ liczba krawędzi. Zostosowanie algorytmu znajduje się w rozdziale 4.2.

3. Implementacja

Kod jest tworzony w języku Python w taki sposób, aby działał tak samo w Pythonie 2.7 i Pythonie 3. Przykładowa sesja interaktywna pokazuje podstawowe operacje na mapach. Warto zwrócić uwagę, że dla grafów abstrakcyjnych krawędzie są instancjami klasy Edge z modułu edges, a wierzchołki są zwykle liczbami lub stringami (muszą być hashowalne i mieć zdefiniowany porządek). W przypadku map krawędzie są instancjami klasy Segment z modułu segments, a wierzchołki instancjami klasy Point z modułu points. W klasie Segments za pomocą dekoratorów **property** zapewniono zgodność z interfejsem klasy Edge.

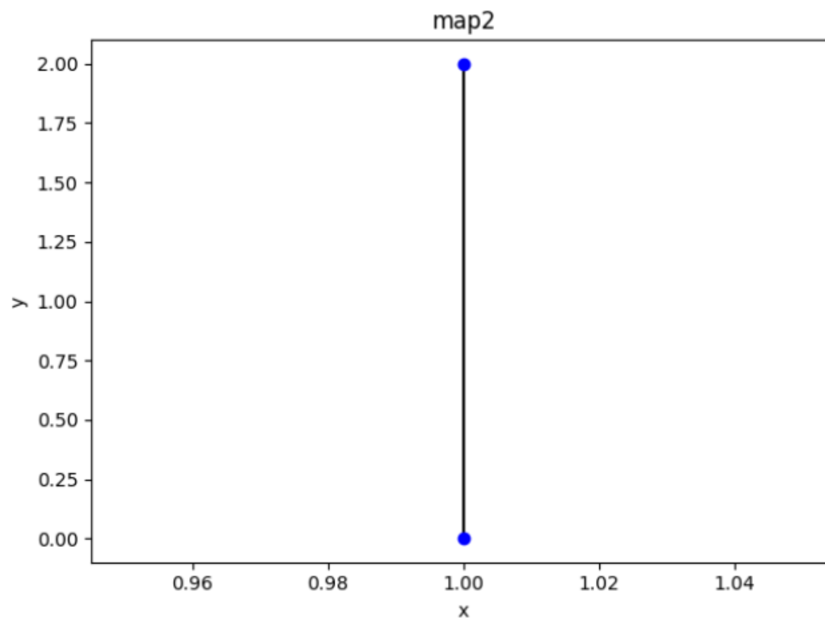
Listing 3.1. Sesja interaktywna z odcinkami prostopadłymi.

```
>>> from points import Point
>>> from segments import Segment
>>> from graphs import Graph
>>> map1 = Graph()
>>> edge_map1 = Segment(Point(0, 1), Point(2, 1))
>>> map1.add_first_edge(edge_map1)
>>> map1.show()
Point(0, 1) : Point(2, 1)(2.0)
Point(2, 1) : Point(0, 1)(2.0)
>>> assert map1.v() == 2 and map1.e() == 1
>>> map1.draw_map("map1")
```

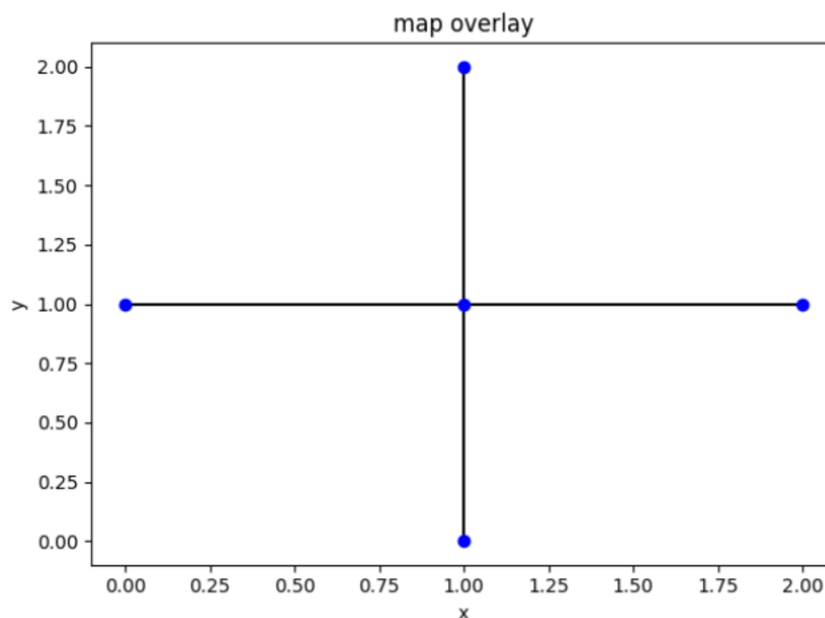


13:1899413399

```
>>> map2 = Graph()
>>> edge_map2 = Segment(Point(1, 0), Point(1, 2))
>>> map2.add_first_edge(edge_map2)
>>> map2.show()
Point(1, 2) : Point(1, 0)(2.0)
Point(1, 0) : Point(1, 2)(2.0)
>>> assert map2.v() == 2 and map2.e() == 1
>>> map2.draw_map("map2")
```




```
>>> map3 = map1.map_overlay(map2)
>>> assert map3.v() == 5 and map3.e() == 4
>>> assert len(list(map3.iterfaces())) == 1
>>> map3.draw_map("map overlay")
```



14:3053450838

3.1. Nowe elementy interfejsu grafów

Podczas pracy nad problemem nakładania map okazało się, że przydatne jest wzbogacenie interfejsu grafów o nowe elementy.

Metoda Graph.iteredges_connected() generuje krawędzie grafu w taki sposób, że wygenerowane krawędzie na każdym etapie tworzą graf spójny, czyli kolejne krawędzie mają co najmniej jeden wspólny wierzchołek z krawędziami wygenerowanymi wcześniej. Wewnętrznie wykonywany jest zmodyfikowany algorytm BFS, który startuje od wierzchołka początkowego krawędzi start_edge, która jest argumentem metody. Krawędź start_edge jest emitowana jako pierwsza przez tą metodę. Metoda jest również używana przy kolorowaniu krawędzi grafu w algorytmie ConnectedSequentialEdgeColoring1. Algorytm generowania krawędzi w sposób spójny został opisany w rozdziale 4.2.

Klasa Graph służy przede wszystkim do przechowywania grafu abstrakcyjnego, czyli zbioru wierzchołków i połączeń między nimi. Klasa ma ponadto atrybuty służące do przechowywania grafu planarnego topologicznego spójnego, gdzie ustalona jest kolejność sąsiadów każdego wierzchołka. Dwa słowniki edge_next i edge_prev realizują strukturę podwójnie połączonej listy krawędzi. Te dwa słowniki pozwalają określić wszystkie ściany grafu planarnego, pozwalają przykładowo obiegać każdą ścianę mając wnętrze ściany po prawej stronie. Z drugiej strony te dwa słowniki to jeszcze za mało, aby szybko określić do której ściany należy dana krawędź skierowana.

Aby poprawić operowanie ścianami dodano dwa nowe słowniki face2edge i edge2face. Słownik edge2face dla każdej krawędzi skierowanej zwraca numer ściany, do której krawędź należy. Ściany są oznaczone liczbami całkowitymi nieujemnymi. Słownik face2edge dla każdej ściany zwraca pewną krawędź, która do niej należy. Zauważmy, że **len**(face2edge) jest równe liczbie ścian grafu planarnego.

Nowe słowniki wymusiły zmiany w metodzie Graph.copy() służącej do kopiowania grafów. Obok istniejącej metody Graph.iterfaces() dodano nową metodę Graph.iterface() do generowania kolejnych krawędzi konkretnej ściany, zaczynając od podanej krawędzi.

Do podanych zmian w opisie grafów planarnych topologicznych dostosowano także moduł planarfactory, który zawiera generatory grafu cyklicznego C_n i grafu koła W_n w postaci grafów topologicznych.

Podczas tworzenia algorytmu nakładania map została dodana nowa metoda prywatna Graph.__edge_of_intersection(), która zwraca krawędź przecięcia dwóch wejściowych grafów. Przechodząc po krawędziach grafu podanego w argumencie (za pomocą metody Graph.iteredges()) dla każdej z nich sprawdzamy, czy występuje przecięcie z **self**. Jeżeli edge.intersect(edge1) jest prawdziwe, zwracamy bieżącą krawędź. W przeciwnym przypadku rzucany jest wyjątek ValueError("edges don't intersect"), który jest sprawdzeniem, czy jako argument nie zostały podane grafy planarne bez punktu przecięcia na płaszczyźnie.

15:1261912896

Kolejna metoda Graph.gen_connected_edges() wykorzystywana w algorytmie za pomocą wyrażenia listowego (ang. *list comprehension*) [9] zwraca wszystkie krawędzie mapy other, które przecinają podaną krawędź edge_main (jest to w tym przypadku bieżąca krawędź algorytmu edge_map2).

Nakładając mapę, dla dobrego podziału krawędzi, musimy zbadać położenie jej wierzchołków oraz położenie wierzchołków innej mapy. W sytuacji, gdy krawędzie są równoległe względem siebie oraz mają jeden wspólny wierzchołek, musimy sprawdzać położenie pozostałych wierzchołków. Stanowią one różnice symetryczną zbiorów wierzchołków obu map. Do pomocy została stworzona metoda Graph.__division(). Przyjmuje ona następujące argumenty: (1) **self** jest mapą, krawędzie której dzielimy w punktach przecięcia; (2) edge_map1 to bieżąca krawędź **self**, w iteracji, w której metoda zostaje wywołana; (3) set_edges_map1 zbiór wierzchołków tej krawędzi; (4) edge_map2 to aktualna krawędź drugiej mapy; (5) points to są wierzchołki mapy pierwszej i drugiej, które są różnicą symetryczną zbioru; (6) divide_points to zbiór punktów podziału drugiej mapy.

Na początku badamy do której mapy należy wierzchołek ze zbioru points (nie jest on wspólnym wierzchołkiem). Jeżeli należy on do zbioru wierzchołków pierwszej mapy set_edges_map1 - oznacza to, że należy on do krawędzi tej mapy. Dalej sprawdzzmy którą krawędź dzieli wierzchołek. Jeżeli wierzchołek drugiej mapy należy do krawędzi grafu **self**, możemy taką krawędź podzielić od razu. Natomiast w przypadku, gdy należy on do krawędzi drugiego grafu, dodajemy ten punkt do zbioru divide_points. Jeżeli żaden warunek nie został spełniony i podziału nie ma, to metoda zwraca wartość False.

Kolejną metodą, która rozszerzyła klasę Graph na rzecz nakładania map, jest Graph.__vertex_in_edge(). Argumenty to zbiór wierzchołków set_vertices i krawędź edge, należącą do **self**. Metoda przechodzi po wierzchołkach i w każdej iteracji sprawdza, czy podany wierzchołek należy do podanej krawędzi. Gdy wierzchołek należy do krawędzi - stosuje się na nią metodę divide_edge, w ten sposób zostaje ona podzielona.

3.2. Nowe elementy obiektów geometrycznych

Interfejs odcinków wzbogacono o dwie nowe metody Segment.parallel() i Segment.perpendicular(). Metody służą do testowania, czy dane dwa odcinki są odpowiednio równoległe i prostopadłe.

16:8731029186

4. Algorytmy

W tym rozdziale został opisany główny algorytm nakładania map, a także wszystkie niezbędne algorytmy wykorzystywane przy nakładaniu.

4.1. Algorytm sprawdzający przecinanie się map

W głównej pętli algorytmu nakładania map wykorzystany został algorytm, który znajduje pierwszą krawędź mapy other, która przecina się z **self**.

Dane wejściowe: Dwie mapy.

Problem: Poszukiwanie krawędzi mapy, która przecina krawędź innej mapy.

Opis algorytmu: Ten algorytm jest niezbędny do poszukiwania pierwszej krawędzi mapy other, która przecina mapę **self**. Pętla główna przechodzi po krawędziach mapy other i sprawdza możliwość przecięcia z każdą krawędzią mapy **self**.

Metoda Segment.intersect() sprawdza, czy odcinki się przecinają. W przypadku ogólnym jeżeli odcinki nie są współliniowe i orientacja końców odcinka względem drugiego (w obie strony) jest inna, oznacza to, że odcinki się przecinają. A w przypadku szczególnym sprawdzamy kolinearność i czy jeden z odcinków zawiera koniec drugiego.

Metoda zwraca pierwszą znalezioną krawędź mapy other, przecinającą jakąś krawędź mapy **self**. Jeżeli krawędź nie została znaleziona, czyli mapy się nie przecinają, wtedy rzucany jest wyjątek ValueError. Spełnienie Warunku przecinania się map jest konieczne dla prawidłowego działania algorytmu nakładania map.

Listing 4.1. Metoda Graph.__edge_of_intersection().

```
class Graph(dict):
    ...
    def __edge_of_intersection(self, other):
        """Return an edge of the graph which intersects with the other."""
        for edge in other.iteredges():
            if any(edge.intersect(edge1) for edge1 in self.iteredges()):
                return edge
        raise ValueError("edges don't intersect")
```


17:9173390703

4.2. Algorytm generowania krawędzi mapy w sposób spójny

Dane wejściowe: Mapa oraz krawędź początkowa.

Problem: Wygenerować krawędzie w taki sposób, aby mapa po dodaniu każdej kolejnej krawędzi miała postać spójną.

Opis algorytmu: Na początku algorytmu umieszczona została walidacja danych wejściowych. Jeżeli graf self jest skierowany albo podana krawędź początkowa start_edge nie należy do mapy self, wtedy dana metoda wyrzuca wyjątek ValueError.

Jeżeli punkt początkowy krawędzi source jest większy od punktu końcowego target, wtedy odwracamy krawędź. Chodzi tutaj o zachowanie konwencji z generatora krawędzi Graph.iteredges(), gdzie krawędzie zawsze mają koniec większy od początku, w sensie porządku w zbiorze wierzchołków.

Algorytm jest stworzony na podstawie algorytmu BFS (ang. *breadth-first search*) [10]. Przechodzenie grafu zaczyna się od wierzchołka początkowego krawędzi start_edge podanej w argumencie metody (po ewentualnym odwróceniu) i polega na przechodzeniu przez wszystkie wierzchołki, do których można dojść od podanego wierzchołka początkowego. W ten sposób algorytm generuje mapę, która po każdej iteracji ma postać spójną. Wykorzystany został słownik parent, który przechowuje punkt, z którego po raz pierwszy doszliśmy do danego wierzchołka, oraz kolejkę wierzchołków krawędzi, które sprawdzamy.

Listing 4.2. Metoda Graph.iteredges_connected().

```
class Graph(dict):
    ...
    def iteredges_connected(self, start_edge):
        """Generate all connected edges from the graph on demand."""
        if self.is_directed():
            raise ValueError("the graph is directed")
        if not self.has_edge(start_edge):
            raise ValueError("edge not in the graph")
        if start_edge.source > start_edge.target:
            start_edge = ~start_edge
        used = set()   # for yielded edges
        parent = dict()   # for BFS tree
        parent[start_edge.source] = None
        parent[start_edge.target] = start_edge.source
        Q = Queue()
        Q.put(start_edge.source)
        Q.put(start_edge.target)
        used.add(start_edge)
        yield start_edge
        while not Q.empty():
            source = Q.get()
            for edge in self.iteroutedges(source):
                if edge.target not in parent:
                    parent[edge.target] = source   # before Q.put
```


18:8071308820

```
        Q.put(edge.target)
    if edge.source > edge.target:
        edge = ~edge
    if edge not in used:
        used.add(edge)
        yield edge
```

4.3. Algorytm dodawania cięciwy do mapy

Dane wejściowe: Mapa, do której zostanie dodana cięciwa.

Problem: Dodanie cięciwy do mapy.

Opis algorytmu: Na początku algorytmu umieszczona została walidacja danych wejściowych. Sprawdzamy, czy dana mapa posiada wierzchołki krawędzi, po czym dodajemy krawędź do mapy. Po jej dodaniu sprawdzamy miejsce, do którego krawędź zostanie dołączona.

Jeżeli początkowy wierzchołek jest stopnia 2, to iterujemy krawędzie wychodzące z wierzchołka edge.source i kiedy wierzchołek bieżącej krawędzi wychodzącej edge2 nie jest równy edge.target przerywamy pętle, po czym aktualizujemy jego położenie za pomocą metody Graph._update3(), która jest stworzona dla struktury wierzchołka stopnia większego niż jeden.

Metoda polega na opisie struktury mapy za pomocą parametrów edge_next[...] oraz edge_prev[...]. Parametr odpowiadający za ścianę to edge2face. W tym przypadku topologia mapy będzie następująca:

```
edge_next[edge2] = edge
edge_next[edge] = edge2
edge_prev[edge2] = edge
edge_prev[edge] = edge2
face1 = self.edge2face[edge2]
```

W przypadku, gdy stopień wierzchołka jest większy niż 2, wtedy znajdujemy edge4 i edge5 wychodzące z edge.source, następnie umieszczamy krawędź edge pomiędzy nimi za pomocą metody Graph._update3() oraz dodajemy zmienną face1 = **self**.edge2face[edge5].

Następnie sprawdzamy stopień wierzchółka edge.target. Jeżeli on wynosi 2, wtedy iterujemy krawędzie wychodzące z wierzchołka edge.target i kiedy wierzchołek bieżącej krawędzi wychodzącej edge3 nie jest równy edge.source przerywamy pętle, po czym aktualizujemy jego położenie za pomocą metody Graph._update3(). Topologia mapy będzie następująca:

```
edge_next[edge3] = ~edge
edge_next[~edge] = edge3
edge_prev[edge3] = ~edge
edge_prev[~edge] = edge3
```

Jeżeli stopień wierzchołka edge.target jest większy niż 2, wtedy znajdujemy edge4 i edge5 wychodzące z edge.target oraz aktualizujemy strukturę mapy:

19:1878960656

```
        edge_next[edge4] = ~edge
        edge_next[~edge] = edge5
        edge_prev[edge5] = ~edge
        edge_prev[~edge] = edge4
```

Końcowym działaniem będzie aktualizacja ścian, gdzie face1 to stary numer, a face2 to nowy numer. Jedna ściana zostanie podzielona na dwie. edge2face dla edge jest teraz face1, a wartości face2edge[face1] = edge oraz face2edge[face2] = ~edge zostają uaktualnione. Po tym przypisujemy nowy numer ściany dla ~edge.

Listing 4.3. Metoda Graph.add_chord().

```
class Graph(dict):
...
    def add_chord(self, edge):
        """Add edge (chord)."""
        assert self.has_node(edge.source)
        assert self.has_node(edge.target)
        self.add_edge(edge)
        if self.degree(edge.source) == 2:
            for edge2 in self.iteroutedges(edge.source):
                if edge2.target != edge.target:
                    break
            self._update3(edge2, edge, edge2)
            face1 = self.edge2face[edge2]
        else:
            edge4, edge5 = self._locate(edge)
            self._update3(edge4, edge, edge5)
            face1 = self.edge2face[edge5]
        if self.degree(edge.target) == 2:
            for edge3 in self.iteroutedges(edge.target):
                if edge3.target != edge.source:
                    break
            self._update3(edge3, ~edge, edge3)
        else:
            edge4, edge5 = self._locate(~edge)
            self._update3(edge4, ~edge, edge5)
        face2 = max(self.face2edge) + 1
        self.edge2face[edge] = face1
        self.face2edge[face1] = edge
        self.face2edge[face2] = ~edge
        edge1 = ~edge
        while True:
            self.edge2face[edge1] = face2
            edge1 = self.edge_next[~edge1]
            if edge1 == ~edge:
                break
```

4.4. Algorytm dodawania liścia do mapy

Dane wejściowe: Mapa, do której zostanie dodany liść.

Problem: Dodanie liścia do mapy.

20:3357729037

Opis algorytmu: Na początku algorytmu sprawdzamy warunek, czy do mapy należy tylko jeden wierzchołek krawędzi podanej w argumencie metody. Następnie krawędź zostaje dodana do mapy. Potem sprawdzamy miejsce, do którego krawędź zostanie dołączona. Jeżeli początkowy wierzchołek jest stopnia 2, to iterujemy krawędzie wychodzące z wierzchołka edge.source i kiedy wierzchołek bieżącej krawędzi wychodzącej edge2 nie jest równy edge.target przerywamy pętle. Po tym aktualizujemy strukturę edge oraz strukturę edge2 (wskazujemy edge_next[...] i edge_prev[...]). Po aktualizacji struktury ma miejsce aktualizacja ścian mapy.

Jeżeli stopień wierzchołka edge2.source jest większy niż dwa, wtedy znajdujemy edge4 i edge5 wychodzące z edge.source. Aktualizacja struktury krawędzi liścia wygląda tak samo, jak w poprzednim przypadku, natomiast później wierzchołek edge jest wstawiony pomiędzy edge4 a edge5 za pomocą edge_next i edge_prev.

Listing 4.4. Metoda Graph.add_leaf().

```
class Graph(dict):
    ...
    def add_leaf(self, edge):
        """Add edge (leaf)."""
        if self.has_node(edge.target):
            edge = ~edge
        assert self.has_node(edge.source)
        assert not self.has_node(edge.target)
        self.add_edge(edge)
        if self.degree(edge.source) == 2:
            for edge2 in self.iteroutedges(edge.source):
                if edge2.target != edge.target:
                    break
            self._update1(~edge)
            self._update3(edge2, edge, edge2)
            face = self.edge2face[edge2]
            self.edge2face[edge] = face
            self.edge2face[~edge] = face
        else:
            edge4, edge5 = self._locate(edge)
            self._update1(~edge)
            self._update3(edge4, edge, edge5)
            face = self.edge2face[edge5]
            self.edge2face[edge] = face
            self.edge2face[~edge] = face
```

4.5. Algorytm nakładania map

Algorytm nakładania map przy wejściu otrzymuje dwie mapy, które nakładają się na siebie na płaszczyźnie, ale które nie są wcale powiązane ze sobą w jakikolwiek sposób w kodzie. Wynikiem końcowym jest nowa mapa, reprezentująca dwie nałożone na siebie mapy wejściowe.

Dane wejściowe: Dwie mapy.

21:2907288374

Problem: Nakładanie dwóch map, przecinających się na płaszczyźnie.

Opis algorytmu: Algorytm rozpoczynamy od tworzenia kopii map podanych jako argumenty metody - **self** i other, ponieważ chcemy uniknąć modyfikacji danych wejściowych w pętli. Na podstawie kopii pierwszej mapy map1, nakładając map2, otrzymujemy mapę wyjściową.

Wykorzystując metodę Graph.__edge_of_intersection(), opisaną w sekcji 4.1, i Graph.iteredges(), opisaną w rozdziale 3.1, przechodzimy po krawędziach mapy map2 i, za pomocą Segment.intersect(), sprawdzamy, czy aktualny odcinek przecina krawędź map1. Po znalezieniu takiej krawędzi zapisujemy ją jako edge_of_intersection.

Glówna pętla: W pętli głównej przechodzimy po wszystkich krawędziach map2 w taki sposób, aby one były ze sobą połączone przy każdej iteracji, tworząc w ten sposób mapę spójną. Dla tej operacji został wykorzystany algorytm opisany w sekcji 4.2. Na stos connected_edges_map1 dodajemy wszystkie krawędzie z map1, które się przecinają z główną, bieżącą krawędzią edge_map2. Do znalezienia takich krawędzi służy metoda gen_connected_edges opisana w sekcji 3.1.

Stworzony był także zbiór punktów dzielenia krawędzi edge_map2 - divide_points., który gromadzi w sobie wszystkie punkty przecięcia mapy map2, ponieważ nie dzielimy jej krawędzi od razu. Pomaga to zapobiec sytuacji, gdy przy dzieleniu krawędzi powstają dwie kolejne i musimy ponownie sprawdzić ich położenie względem map1.

Kolejnym krokiem jest przechodzenie po krawędziach map2, które przecinają głowną krawędź edge_map2. Przy wchodzeniu do pętli sciągamy ze stosu ostatnio dodaną krawędź edge_map1 i sprawdzamy, czy jest równoległa do edge_map2.

Równoległe krawędzie: Sprawdzamy ile wspólnych punktów mają zbiór wierzchołków krawędzi edge_map1 i krawędzi edge_map2. Jeżeli obie krawędzie mają takie same wierzchołki, wtedy liczność sumy zbiorów wierzchołków wynosi dwa. Wiadomo, że są one do siebie równoległe i ich wierzchołki są takie same, oznacza to, że te krawędzie nakładają się na siebie. W takim przypadku przechodzimy do następnej iteracji pętli **while**.

Gdy krawędzie mają jeden wspólny punkt możliwe są cztery przypadki:

1. Punkt początkowy edge_map1 i punkt początkowy edge_map2 są te same.
2. Punkt początkowy edge_map1 i punkt końcowy edge_map2 są te same.
3. Punkt końcowy edge_map1 i punkt początkowy edge_map2 są te same.
4. Punkt końcowy edge_map1 i punkt końcowy edge_map2 są te same.

W każdej z tych możliwości, musi być sprawdzone położenie pozostałego wierzchołka map1 i map2.

Jeżeli dwie krawędzie są równoległe i mają jeden wspólny wierzchołek, to istnieje następujące położenie pozostałych wierzchołków:

1. Wierzchołek krawędzi edge_map1 leży na edge_map2.
2. Wierzchołek krawędzi edge_map2 leży na edge_map1.
3. Krawędzie nie mają więcej punktów przecięcia. W tym przypadku przechodzimy do kolejnej iteracji pętli.

22:5927381375

Wiadomo, że krawędzie mają jeden wspólny wierzchołek, jest to jeden wspólny element zbioru set_edges_map1 i set_edges_map2 - stanowi on przecięcie zbiorów. Natomiast pozostałe wierzchołki, które musimy zbadać, będą zawierać się w różnicy symetrycznej zbiorów. Tą różnice zapisujemy do edges_nodes_diff i za pomocą metody Graph.__division() sprawdzamy ich położenie.

W metodzie Graph.__division() najpierw sprawdzamy, do której mapy należy wierzchołek, przechowywany w edges_nodes_diff. Nie jest to wiadome, ponieważ zbiór nie jest uporządkowany. W sytuacji, gdy wierzchołek krawędzi mapy map1 należy do krawędzi mapy map2, dodajemy ten wierzchołek do zbioru divide_points, ponieważ w pętli **while** nie zmieniamy mapy map2. Natomiast jeżeli wierzchołek krawędzi mapy map2 należy do krawędzi mapy map1, wtedy możemy podzielić krawędź map1 od razu.

W przypadku, kiedy krawędzie nie mają wspólnych wierzchołków, poniższe możliwości:

1. Kiedy krawędź map2 zawiera się w edge_map1. Dzielimy krawędź map1, dodając do niej wierzchołek edge_map2.source. Następnie za pomocą metody Graph.iteroutedges() generujemy krawędzie, wychodzące z edge_map2.source i przydielamy je do edge1copy1, edge1copy2. Potrzebujemy te dwie części dla tego, żeby zlokalizować do której należy inny wierzchołek mapy map2 - edge_map2.target. Tą część, do której on należy, odpowiednio dzielimy.
2. Kiedy wierzchołek edge_map1.source zawiera się w krawędzi mapy map2. Wtedy ten wierzchołek zostaje dodany do zbioru divide_points, po czym sprawdzamy, czy w edge_map2 zawiera się kolejny wierzchołek edge_map1.target. Jeżeli to prawda - też dodajemy go do zbioru. W przeciwnym przypadku sprawdzamy, czy któryś z wierzchołków krawędzi edge_map2 zawiera się w edge_map1 i, jeżeli tak, dzielimy krawędź w tym punkcie. Za to opdowiada metoda Graph.__vertex_in_edge(), przyjmująca zbiór wierzchołków i krawędź. Sprawdzane są wierzchołki po kolei, jeśli któryś z nich zamiera się w podanej krawędzi mapy map1, to w metodzie dzielimy krawędź.
3. Kiedy wierzchołek edge_map1.target zawiera się w krawędzi mapy map2. Wtedy ten wierzchołek zostaje dodany do zbioru divide_points i wywołana jest metoda Graph.__vertex_in_edge(), sprawdzająca wierzchołki edge_map2 w krawędzi edge_map1.

Przykłady uruchamiania algorytmu dla map mających równoległe krawędzie są w rozdziale B.2. Następnym przypadkiem szczególnym są nierównoległe krawędzie.

Krawędzie nie są równoległe względem siebie: Jeżeli krawędzie nie są równoległe i mają jeden wspólny wierzchołek - przechodzimy do następnej iteracji, ponieważ na pewno nie mają więcej punków przecięć. W innych przypadkach sprawdzamy możliwości kiedy:

1. Wierzchołek początkowy edge_map1.source zawiera się w krawędzi mapy map2. Dodajemy go do divide_points.
2. Wierzchołek końcowy edge_map1.target zawiera się w krawędzi mapy map2. Dodajemy go do divide_points.
3. Wierzchołek początkowy edge_map2.source zawiera się w krawędzi mapy map1. Dzielimy krawędź edge_map1 w tym punkcie.
4. Wierzchołek końcowy edge_map2.target zawiera się w krawędzi mapy map1. Dzielimy krawędź edge_map1 w tym punkcie.

23:1039915438

5. Jeżeli żaden z powyższych warunków nie zachodzi. Szukamy punktu przecięcia krawędzi za pomocą metody Segment.intersection_point(), dzielimy w tym punkcie krawędzi z mapy map1, dodajemy ten punkt do divide_points.

Przykłady uruchamiania algorytmu dla map mających równoległe krawędzie są w rozdziale B.3.

Tutaj kończy się przechodzenie po krawędziach mapy map1, połączonych z edge_map1 w pętli **while**. Wszystkie krawędzie mapy map1 zostały podzielone, a punkty przecięcia krawędzi mapy map2 zostały dodane do divide_points.

Pozostała część algorytmu odpowiada za dzielenie krawędzi mapy map2 i dodanie brakujących krawędzi do map1.

Dzielenie krawędzi map2, dodawanie liści i cięciw: Przy dzieleniu mapy map2 został stworzony zbiór krawędzi mapy. Przechodząc przez wszystkie punkty podziału dodane w poprzedniej części algorytmu, sprawdzamy czy zawiera się bieżący punkt w krawędzi i jeżeli tak, to dzielimy krawędź edge_map2, usuwamy krawędź ze zbioru i dodajemy do niego nowo powstałe, przez dzielenie, krawędzie.

Po tym jak wszystkie mapy zostały podzielone, uzupełniamy wyjściową mapę map1. Przechodząc po krawędziach, które powstały w wyniku dzielenia mapy map2, sprawdzamy czy mapa map1 już je ma. Jeżeli tak, to w tym przypadku kończymy pętle. W przeciwnej sytuacji, jeżeli mapa map1 ma już oba wierzchołki krawędzi, do map1 zostanie dodana cięciwa. Jeżeli ma tylko jeden z wierzchołków, do map1 zostanie dodany liść.

Kiedy mapa map1 jest uzupełniona, wtedy zostaje ona zwrócona jako wynik metody.

Złożoność: Złożoność czasowa algorytmu jest dość trudna do oszacowania. Zależy ona nie tylko od parametrów map wejściowych (liczby wierzchołków i krawędzi), ale też od sposobu przecinania się map. Końcowa liczba wierzchołków i krawędzi jest na ogół większa od sumy odpowiednio wierzchołków i krawędzi map wejściowych. Tak więc algorytm nakładania się map jest algorytmem czułym na wyjście (ang. *output-sensitive algorithm*).

24:1195235824

5. Podsumowanie

W ramach pracy został opracowany algorytm nakładający mapy, czyli grafy nieskierowane, spójne i płaskie. Implementacja algorytmu znalazła się głównie w metodzie Graph.map_overlay(), ale pewne etapy znajdują się w metodach pomocniczych. Algorytm wymaga, aby nakładane mapy przecinały się, ponieważ tylko wtedy wynikowa mapa będzie spójna.

Do implementacji w języku Python została wykorzystana podwójnie wiązana lista krawędzi, ponieważ ta struktura danych pozwala przechowywać informację o topologicznej strukturze grafu oraz ułatwia operowanie ścianami mapy.

Algorytm zaczyna budowę wyjściowej mapy od kopii pierwszej mapy wejściowej. Następnie nakładane są kolejne krawędzie drugiej mapy wejściowej, emitowane w sposób spójny. Podczas nakładania krawędzi może nastąpić podział zarówno nakładanej krawędzi, jak i krawędzi nałożonych wcześniej. W końcowym etapie nakładania krawędzi pojawiają się tylko dwa przypadki bazowe: dodawanie liścia lub dodawanie cięciwy.

Wierzchołkami mapy są instancje klasy Point z modułu points. Warto zwrócić uwagę, że jeżeli współrzędne wejściowych wierzchołków będą całkowite lub wymierne (klasa Fraction), to punkty przecięcia będą obliczane dokładnie i również będą liczbami wymiernymi. Pozwala to uniknąć problemów wynikających ze skończonej dokładności obliczeń na liczbach **float**.

Algorytm w przyszlości może być rozszerzony, w celach praktycznego wykorzystania nakładania map. Należałoby dodać pewne atrybuty do wierzchołków i krawędzi map, a następnie określić przenoszenie się atrybutów podczas operacji dzielenia krawędzi i tworzenia nowych wierzchołków.

Następnym pożądanym usprawnieniem w posługiwaniu się mapami byłby interfejs graficzny ułatwiający budowanie map i wizualizację wyników różnych operacji na mapach.

25:5875738036

A. Kod źródłowy

Listing A.1. Metoda Graph.map_overlay().

```
def map_overlay(self, other):
    """―――――――――description―――――――――"""
    map1, map2 = self.copy(), other.copy()
    edge_of_intersection = self.__edge_of_intersection(other)

    for edge_map2 in other.iteredges_connected(edge_of_intersection):
        connected_edges_map1 = map1.gen_connected_edges(edge_map2)
        divide_points = set()
        vertices_edge_map2 = set([edge_map2.source, edge_map2.target])

        while connected_edges_map1:
            edge_map1 = connected_edges_map1.pop()
            vertices_edge_map1 = set([edge_map1.source, edge_map1.target])

            if edge_map1.parallel(edge_map2):
                if len(vertices_edge_map1 & vertices_edge_map2) == 2:
                    continue
                elif len(vertices_edge_map1 & vertices_edge_map2) == 1:
                    edges_vertices_diff = vertices_edge_map1 ^ \
                                            vertices_edge_map2
                    if not map1.__division(edge_map1, edge_map2,
                        vertices_edge_map1, edges_vertices_diff,
                        divide_points):
                        continue
                else:
                    if edge_map2.source in edge_map1 \
                            and edge_map2.target in edge_map1:
                        edge1copy1, edge1copy2 = \
                            map1.iteroutedges(edge_map2.source)
                        if edge_map2.target in edge1copy1:
                            map1.divide_edge(edge1copy1, edge_map2.target)
                        else:
                            map1.divide_edge(edge1copy2, edge_map2.target)
                    elif edge_map1.source in edge_map2:
                        divide_points.add(edge_map1.source)
                        if edge_map1.target in edge_map2:
                            divide_points.add(edge_map1.target)
                        else:
                            map1.__vertex_in_edge(vertices_edge_map2, edge_map1)
                    elif edge_map1.target in edge_map2:
                        divide_points.add(edge_map1.target)
                        map1.__vertex_in_edge(vertices_edge_map2, edge_map1)
                    else:
                        raise ValueError("impossible")
            else:
                if len(vertices_edge_map1 & vertices_edge_map2) == 1:
```


26:5304216504

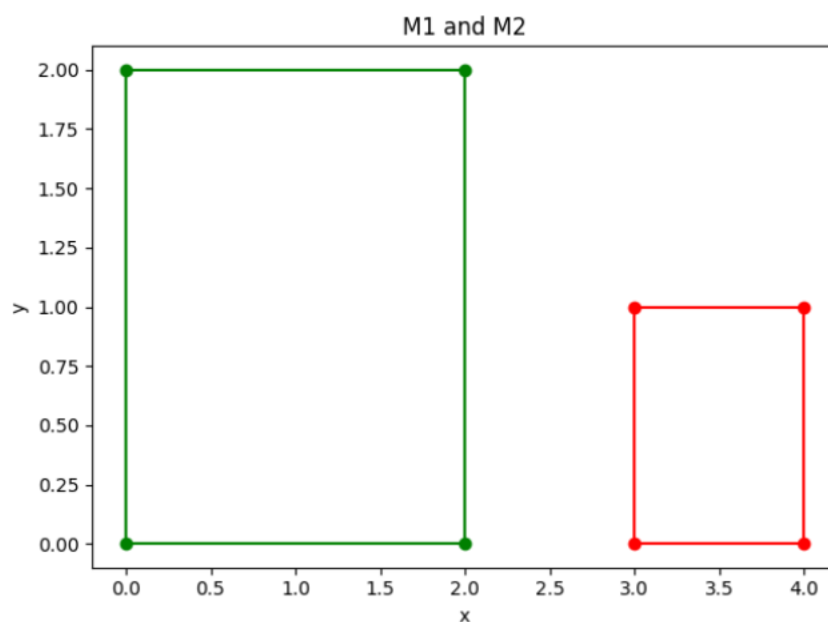
```
                continue
            if edge_map1.source in edge_map2:
                divide_points.add(edge_map1.source)
            elif edge_map1.target in edge_map2:
                divide_points.add(edge_map1.target)
            # map1.__vertex_in_edge(vertices_edge_map2, edge_map1)
            elif edge_map2.source in edge_map1:
                map1.divide_edge(edge_map1, edge_map2.source)
            elif edge_map2.target in edge_map1:
                map1.divide_edge(edge_map1, edge_map2.target)
            else:
                point = edge_map1.intersection_point(edge_map2)
                map1.divide_edge(edge_map1, point)
                divide_points.add(point)

    edge_set = set([edge_map2])
    while divide_points:
        point = divide_points.pop()
        for edge in edge_set:
            if point in edge:
                map2.divide_edge(edge, point)
                edge_set.remove(edge)
                edge_set.update(map2.iteroutedges(point))
                break
    for edge in edge_set:
        if map1.has_edge(edge):
            continue
        elif map1.has_node(edge.source) and map1.has_node(edge.target):
            map1.add_chord(edge)
        elif map1.has_node(edge.source) or map1.has_node(edge.target):
            map1.add_leaf(edge)
        else:
            raise ValueError("impossible")
return map1
```

B. Testy algorytmów

Dodatek ten zawiera testy poprawnościowe oraz wydajnościowe algorytmu nakładania map. Niektóre wyniki są przedstawione w formie graficznej.

B.1. Test map bez przecinających się krawędzi.

Test polega na wykryciu walidacji, czy mapy się w ogóle przecinają. Jeżeli metoda Graph.__edge_of_intersection() (4.1) nie znajdzie krawędzi przecięcia dwóch map, powinna ona wyrzucić wyjątek ValueError("edges don't intersect"). Poniżej podany wynik uruchamiania potwierdza walidacje.



Rysunek B.1. Test 1. Mapy wejściowe bez przecięcia.

Listing B.1. Test dla map bez przecięcia.

```
Traceback (most recent call last):
  File ".../src/main/test_mapoverlay1.py", line 26, in <module>
    M3 = M1.map_overlay(M2)
  File ".../src/main/graphs.py", line 562, in map_overlay
    edge_of_intersection = self.__edge_of_intersection(other)
  File ".../src/main/graphs.py", line 524, in __edge_of_intersection
    raise ValueError("edges don't intersect")
ValueError: edges don't intersect
```


28:6310332187

B.2. Test map z równoległymi krawędziami

Testy z tej sekcji sprawdzają poprawność nakładania map przy różnych możliwych krawędziach równoległych. Na rysunku B.2 są pokazane mapy z nakładającą się krawędzią i dwoma wspólnymi wierzchołkami. W tym przypadku wejściowa mapa na rysunku B.3 nie posiada dodatkowo stworzonych wierzchołków.

Na kolejnym rysunku B.4 mapy wejściowe mają równoległą krawędź z jednym wspólnym wierzchołkiem, krawędź wejściowej mapy została podzielona na dwie części (rysunek B.5).

Natomiast na rysunku B.6 widać, że krawędź jednej mapy zawiera się w innej. W wyniku krawędź jednej mapy zostaje podzielona na trzy części (rysunek B.7).

Kolejna możliwość nakładania się dwóch krawędzi to wtedy, kiedy jedna krawędź zawiera w sobie wierzchołek innej krawędzi i na odwrót, rysunek B.8. Wynik nakładana jest na rysunku B.9, krawędzie obu map zostają podzielone.

Przypadek na rysunku B.10 reprezentuje częsciowe przecinanie się i nakładanie się krawędzi. Wynik nakładania takich map na rysunku B.11 Widzimy nowe wierzchołki utworzone na przecięciu krawędzi.

B.3. Test map z przecinającymi się krawędziami

W tym przypadku są kilka możliwości przecięcia. Jedną z nich to przecięcie krawędzi z jednym wspólnym wierzchołkiem. Takie dane wejściowe są pokazane na rys. B.12, wyjściowa mapa znajduje się na rysunku B.13.

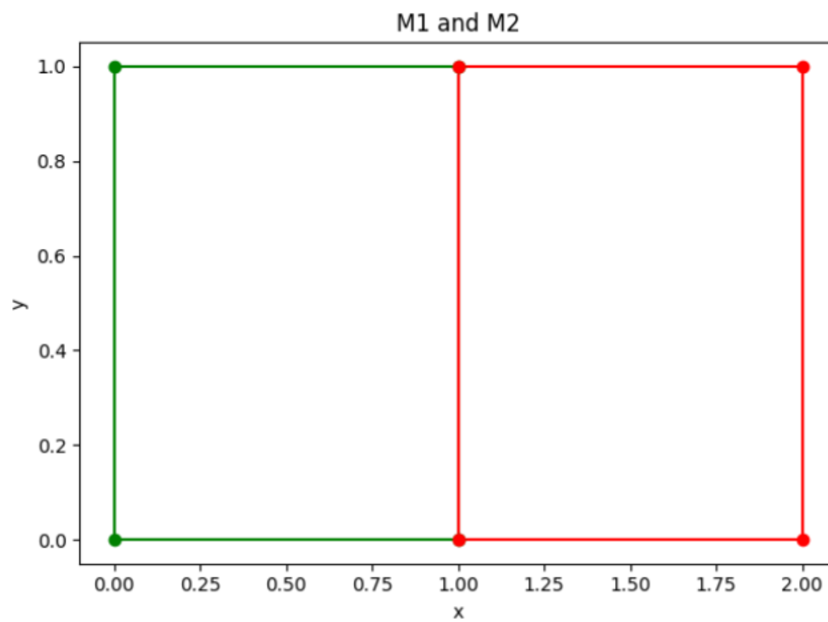
Kolejne możliwe nakładanie prostopadłych krawędzi, bez wspólnych wierzchołków na rysunku B.14 (mapy wejściowe) i B.15 (mapa wyjściowa).

B.4. Test złożoności wykonania algorytmu nakładania map

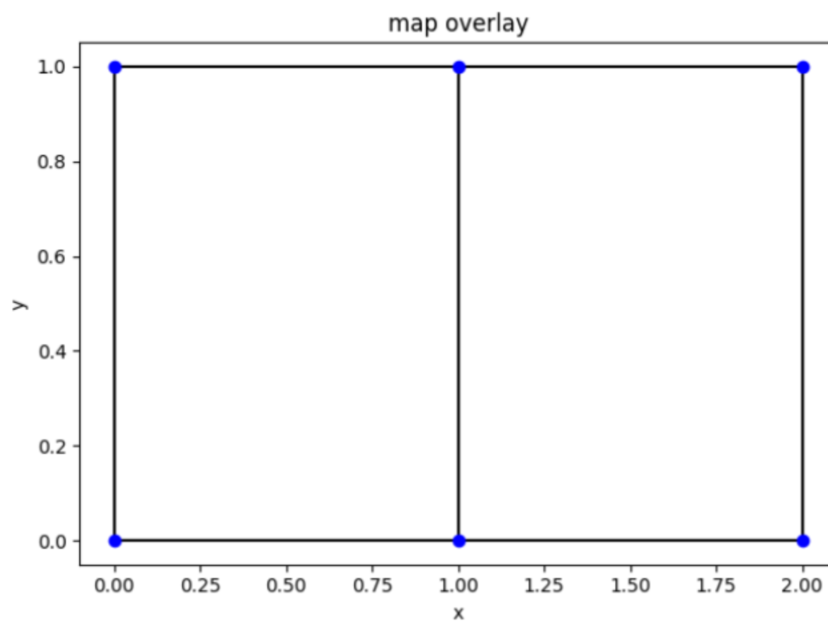
Test został opracowany w celu znalezienia złożoności algorytmu nakładania map. Dla tego testu wykorzystana zostaje metoda, która tworzy mapę z odpowiednią liczbą grzebieni s, podaną jako argument oraz mapę - prostokąt, która przecina grzebienie pierwszej mapy. Po tym stworzone mapy zostają na siebie nałożone. Przykłady:

— mapy wejściowe dla S = 1 na rys. B.16 oraz mapa wyjściowa na rys. B.17,
— mapy wejściowe dla S = 5 na rys. B.18 oraz mapa wyjściowa na rys. B.19,
— mapy wejściowe dla S = 10 na rys. B.20 oraz mapa wyjściowa na rys. B.21,

Dla ilości grzebieni S, ilości krawędzi E_1 oraz wierzchołków V_1 mapy map1 zachodzi $V_1 = E_1 = 4S + 2$. Paramert S też można zapisać za pomocą liczby przecięć map X, $S = 4X$. Dla tego żeby określić przykładową złożoność obliczeniową został dodany wykres na rysunku B.22.

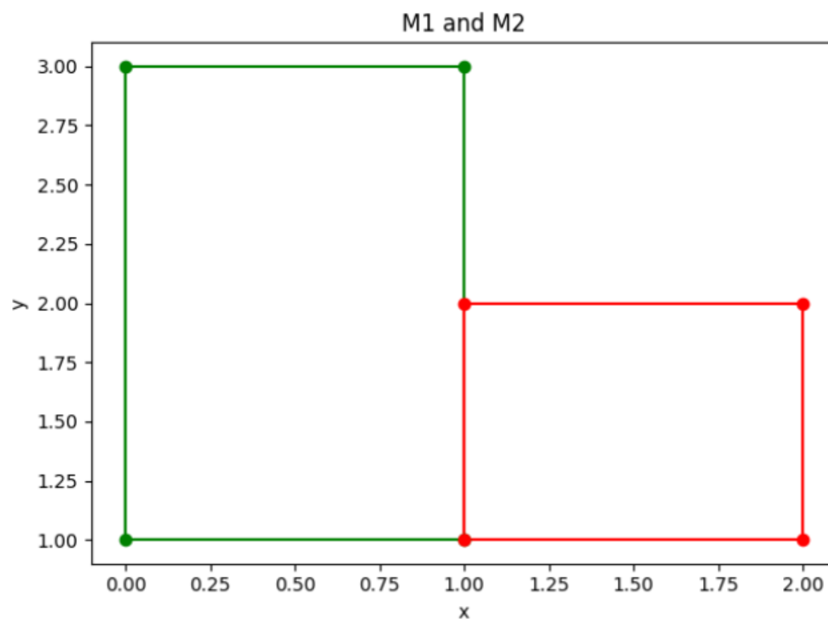


Rysunek B.2. Test2. Mapy wejściowe ze wspólną krawędzią.

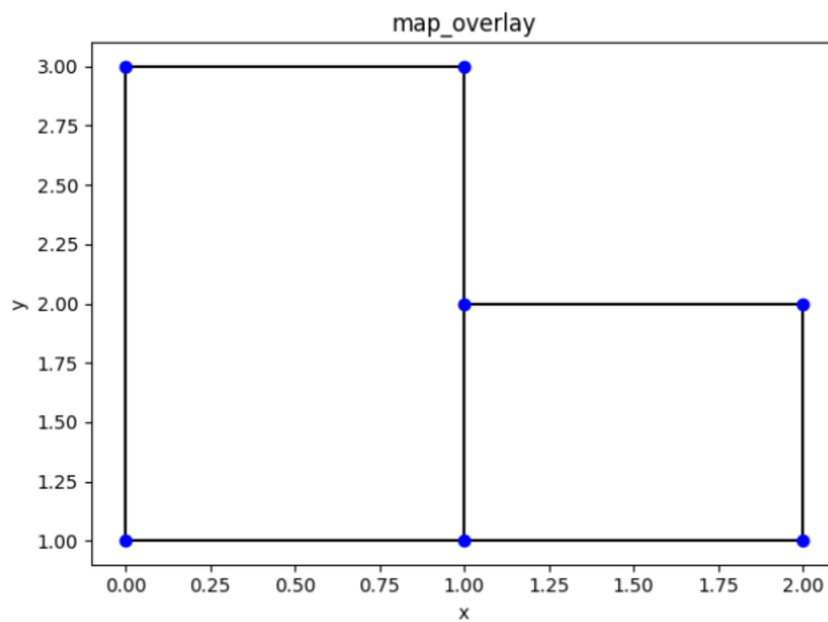


Rysunek B.3. Test2. Mapa wyjściowa.

30:3075415714

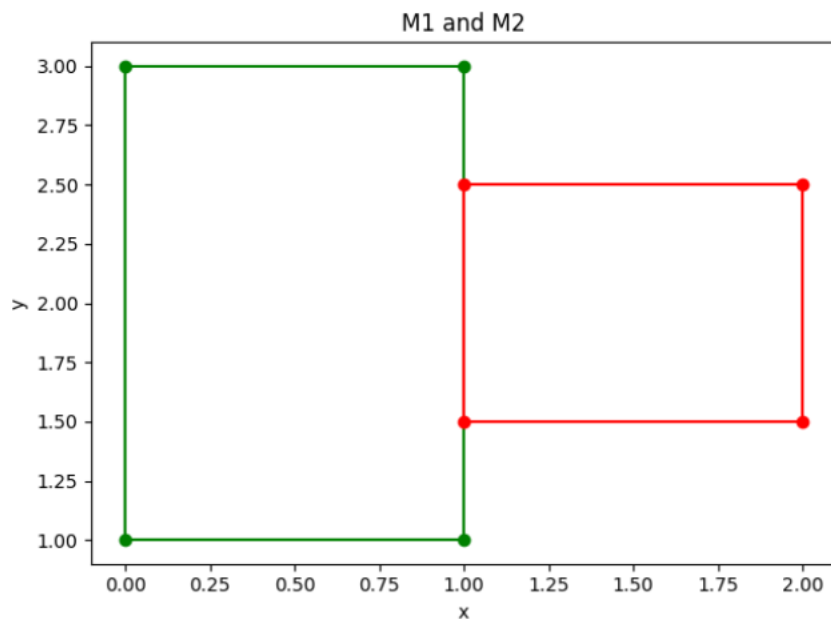


Rysunek B.4. Test3. Mapy wejściowe.

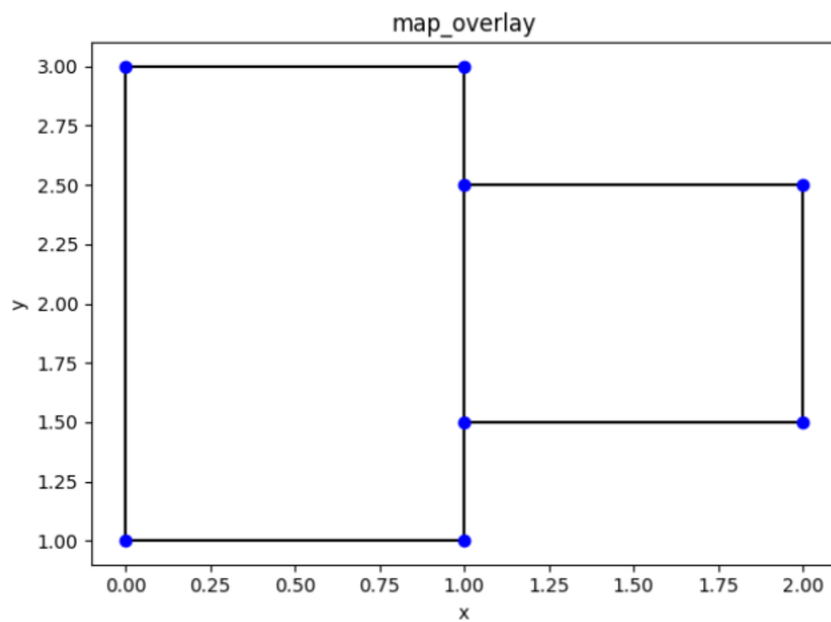


Rysunek B.5. Test3.Mapa wyjściowa.

31:2206634137

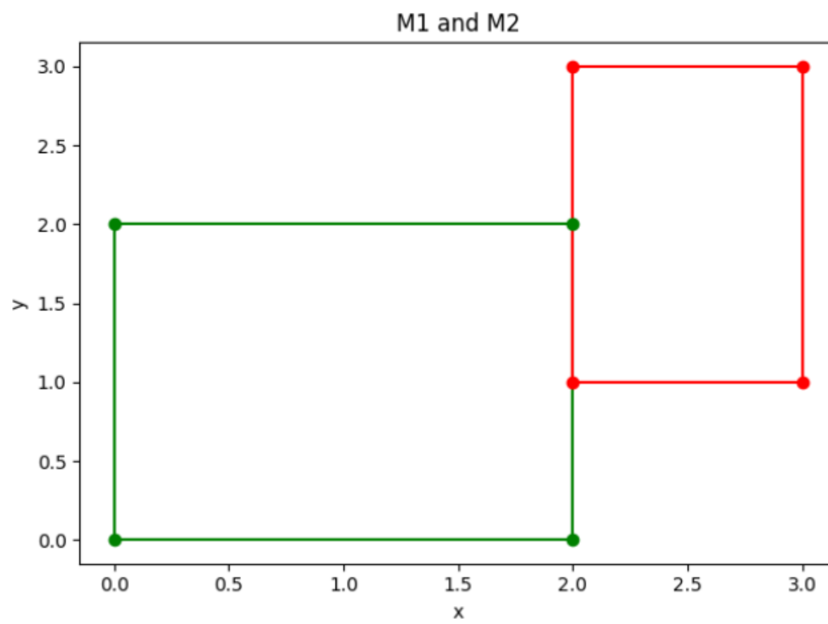


Rysunek B.6. Test4. Mapy wejściowe.

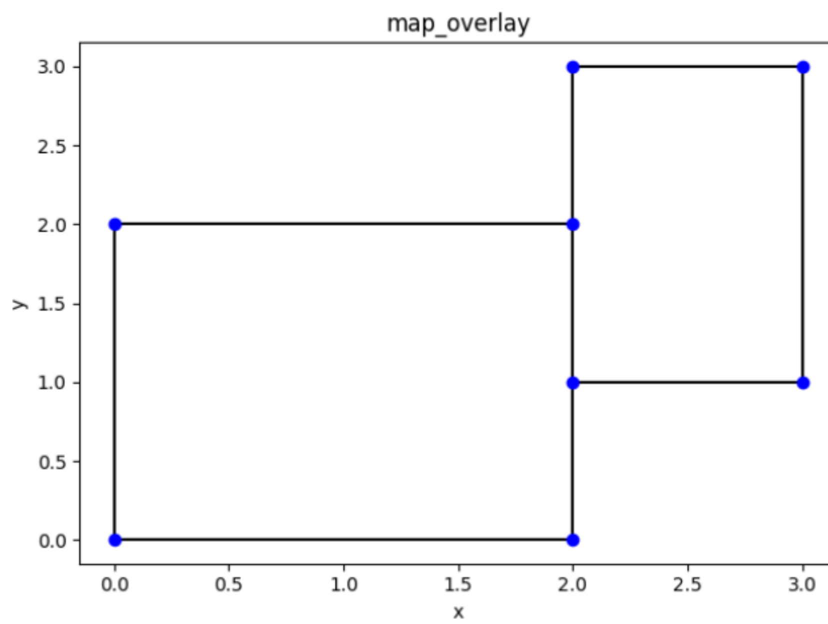


Rysunek B.7. Test4. Mapa wyjściowa.

32:7248877253

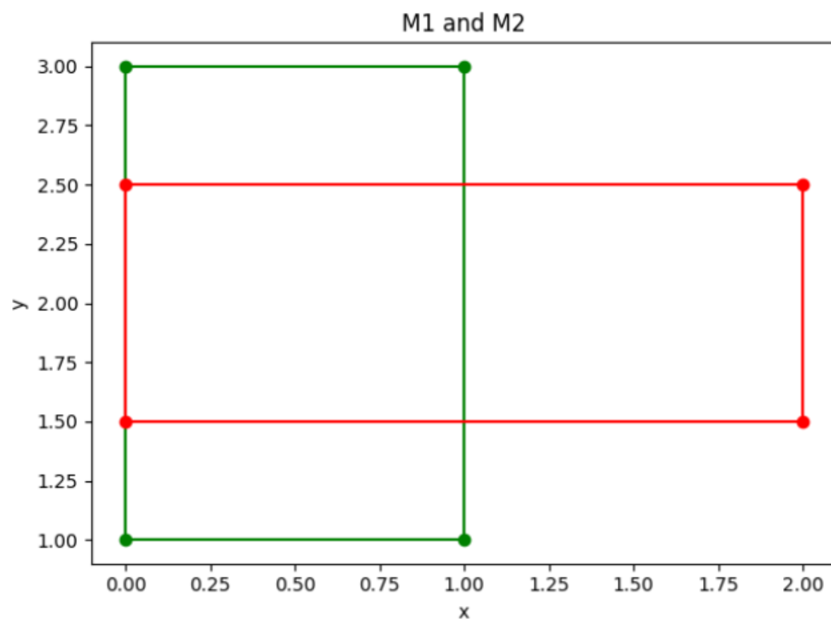


Rysunek B.8. Test5. Mapy wejściowe.

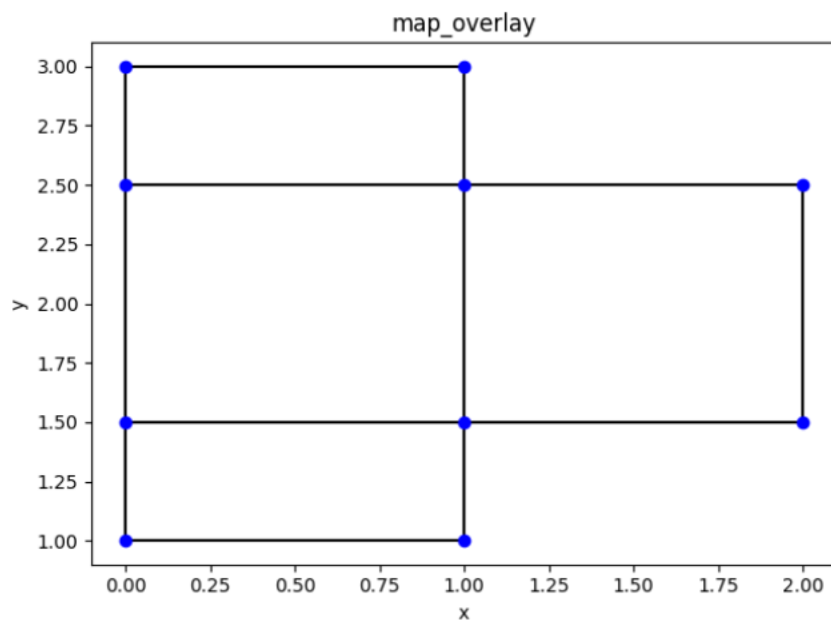


Rysunek B.9. Test5. Mapa wyjściowa.

33:8494751616

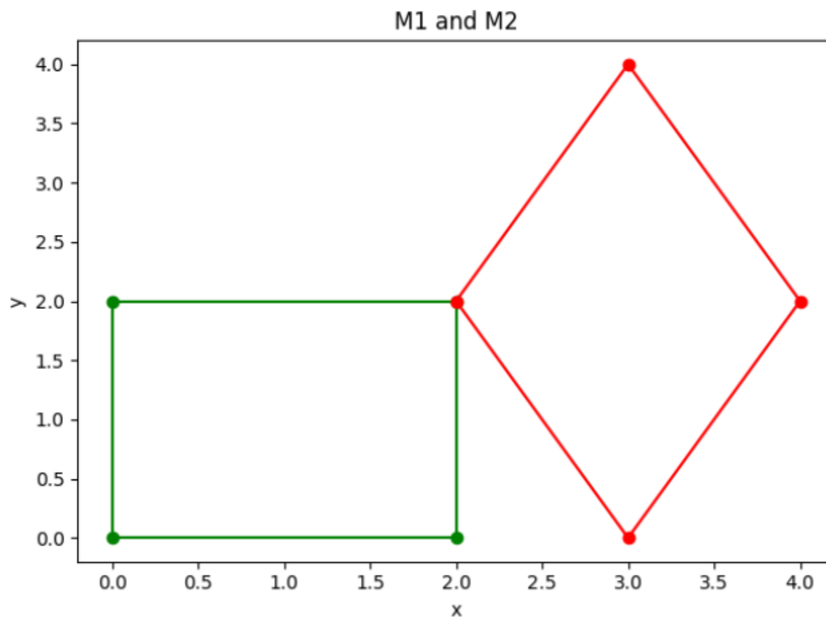


Rysunek B.10. Test6. Mapy wejściowe.

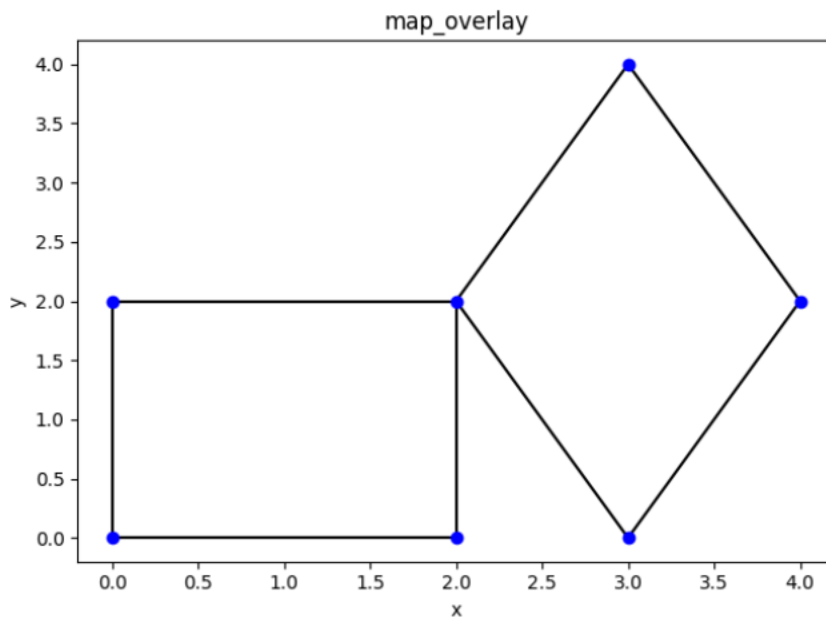


Rysunek B.11. Test6. Mapa wyjściowa.

34:5366085430

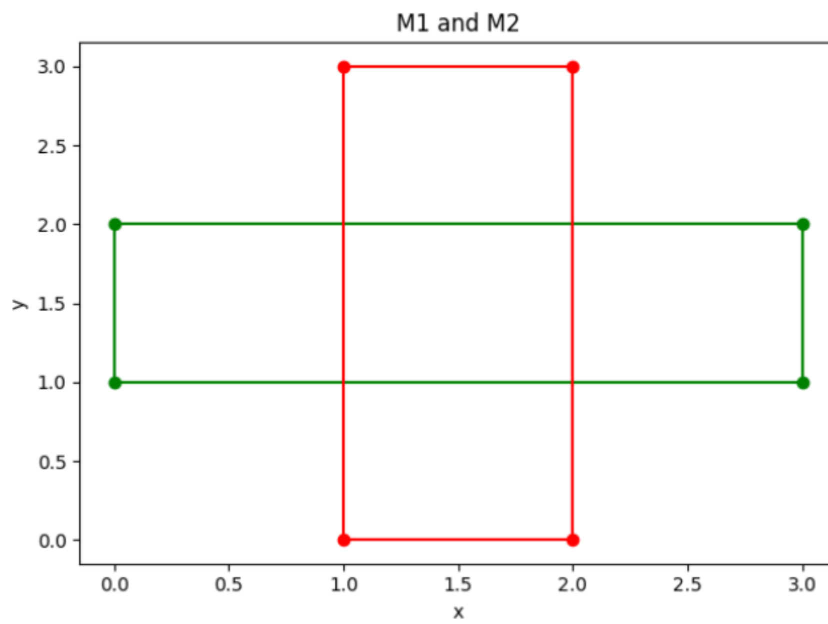


Rysunek B.12. Test7. Mapy wejściowe.

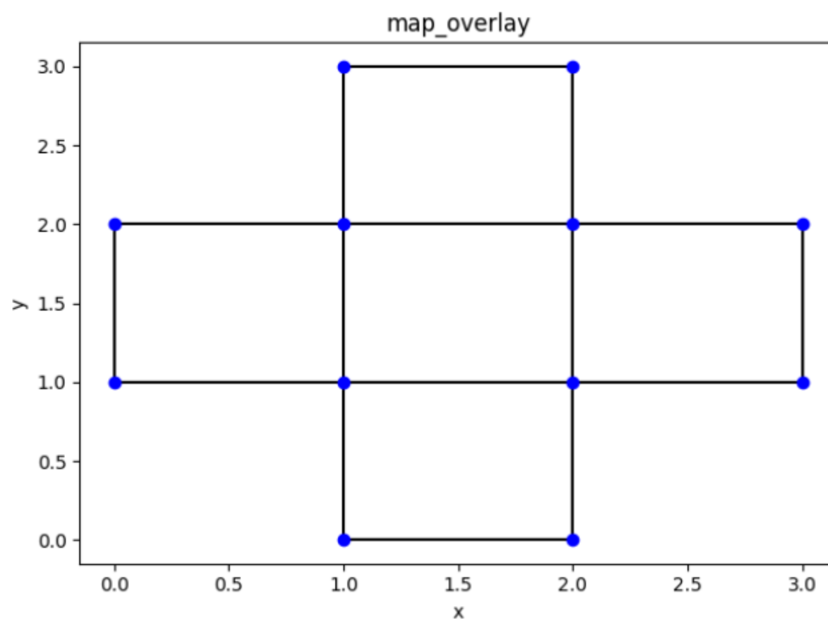


Rysunek B.13. Test7. Mapa wyjściowa.

35:1145061130

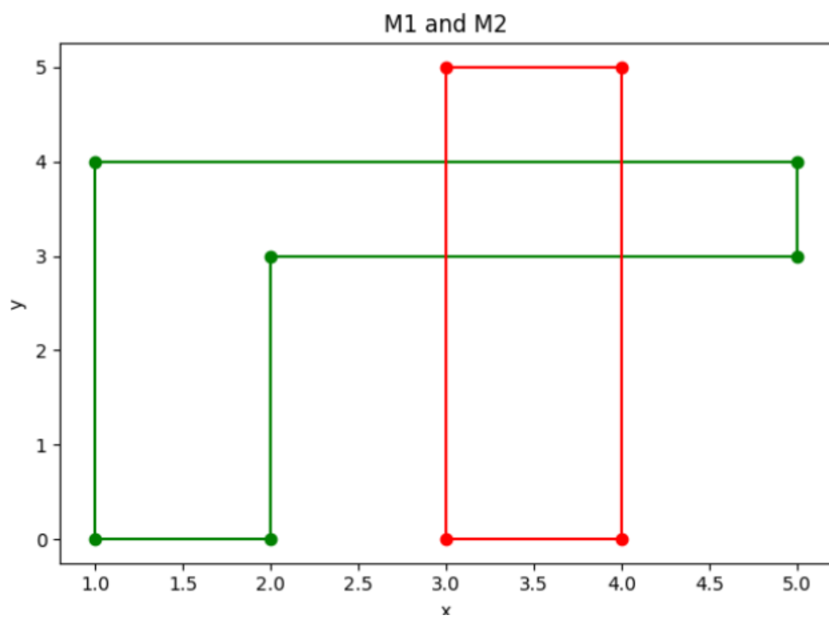


Rysunek B.14. Test8. Mapy wejściowe.

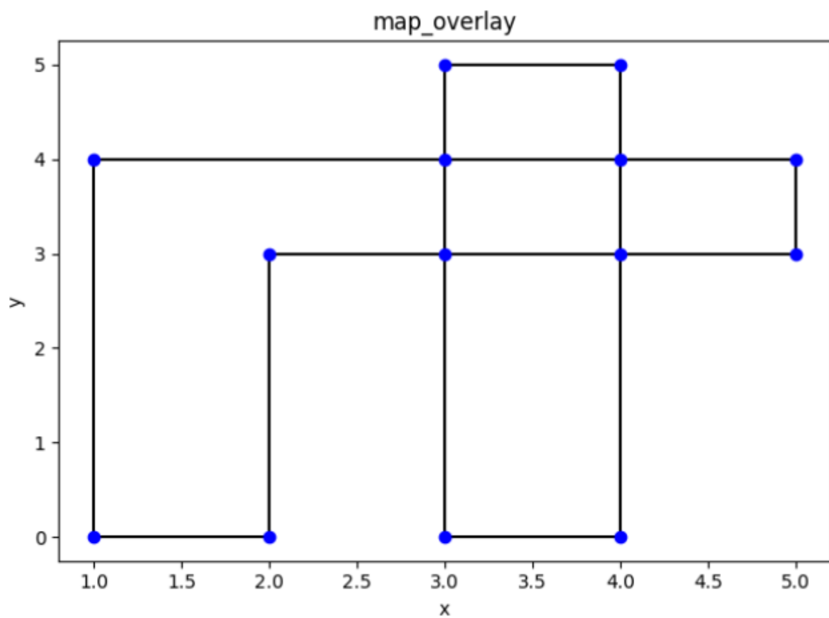


Rysunek B.15. Test8. Mapa wyjściowa.

36:4656708170

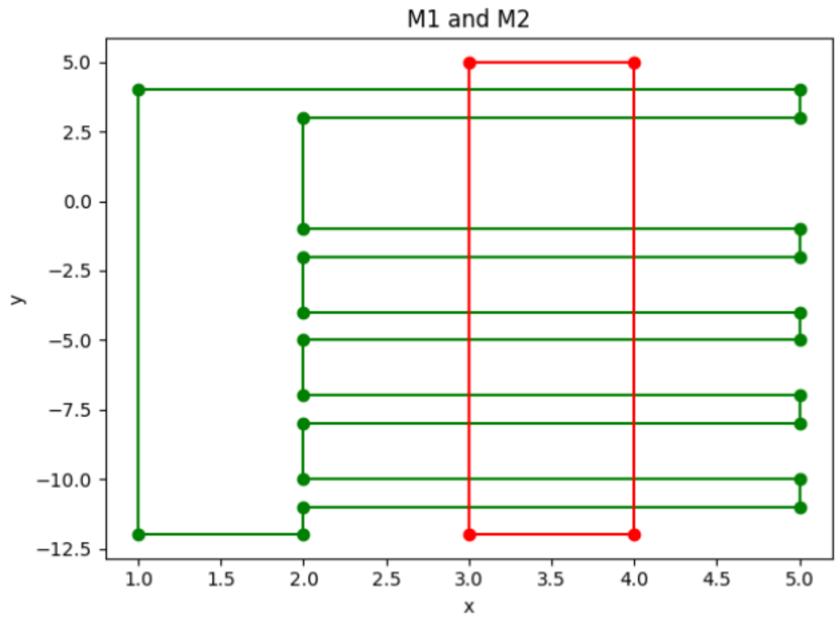


Rysunek B.16. Test9. Mapy wejściowe z parametrem $S = 1$.

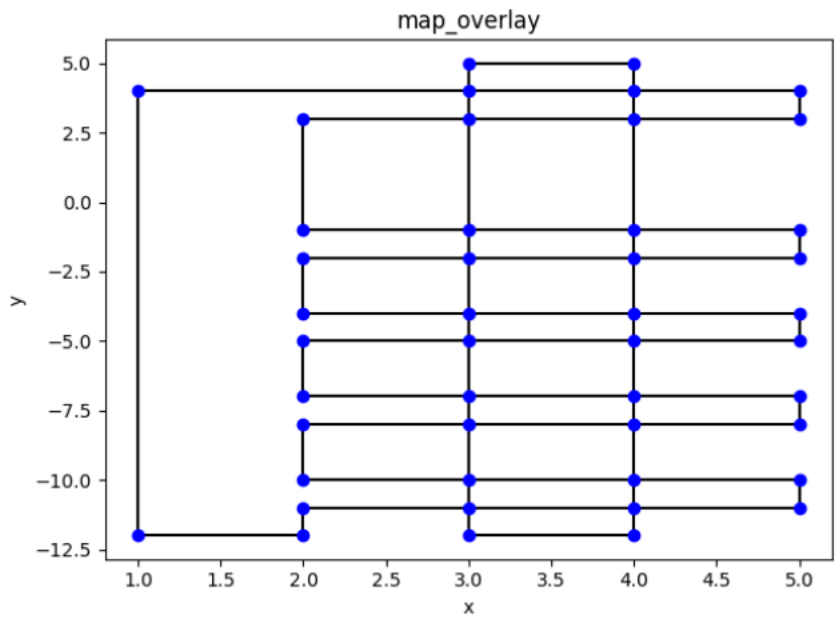


Rysunek B.17. Test9. Mapa wyjściowa z parametrem $S = 1$.

37:8439914652

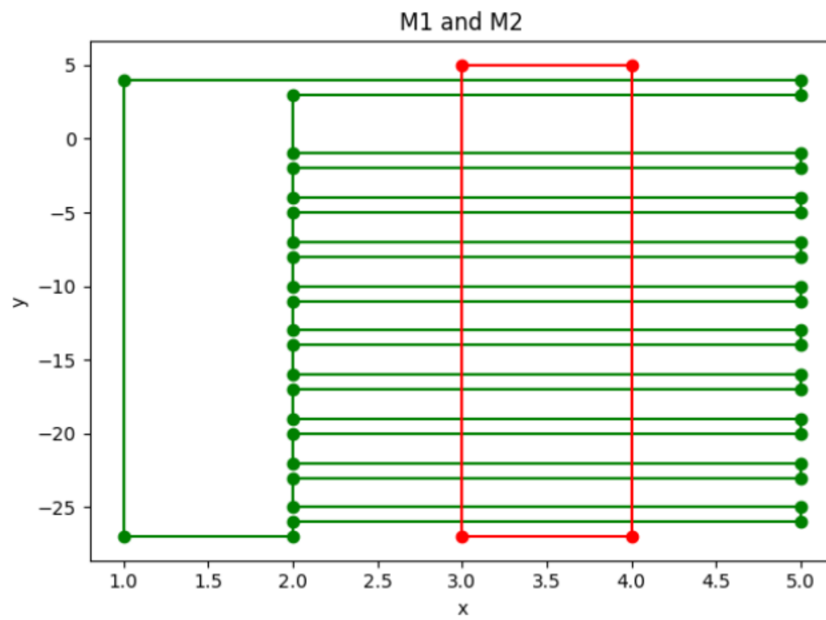


Rysunek B.18. Test9. Mapy wejściowe z parametrem $S = 5$.

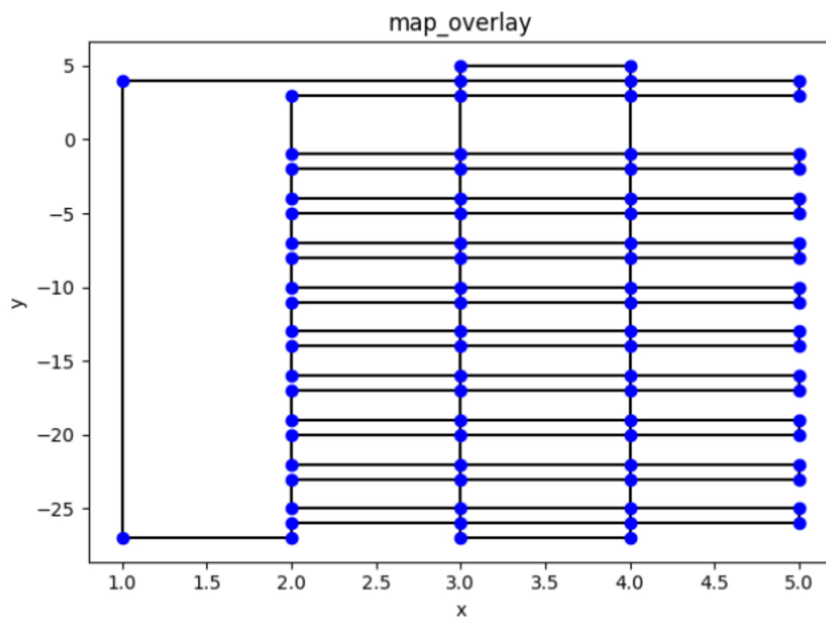


Rysunek B.19. Test9. Mapa wyjściowa z parametrem $S = 5$.

38:8678672424

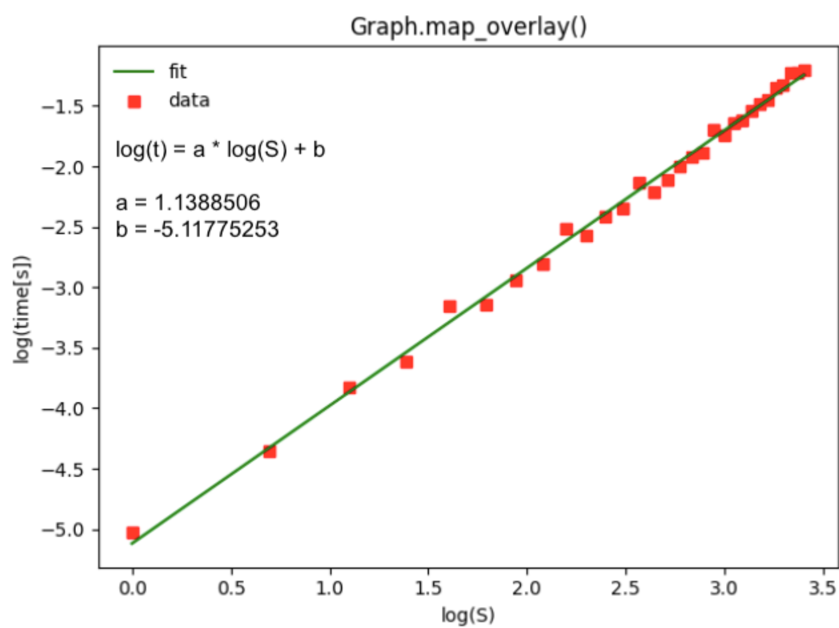


Rysunek B.20. Test9. Mapy wejściowe z parametrem S = 10.



Rysunek B.21. Test9. Mapa wyjściowa z parametrem S = 10.

39:1604893150



Rysunek B.22. Test9. Wykres wydajności nakładania map. Współczynnik a lekko przekraczający 1 sugeruje ponad liniową zależność złożoności obliczeniowej od liczby przecięć.

40:2955286521

Bibliografia

[1] Wikipedia, Computational geometry, 2020,
https://en.wikipedia.org/wiki/Computational_geometry.

[2] Wikipedia, Graph, 2020,
https://en.wikipedia.org/wiki/Graph_(discrete_mathematics).

[3] Wikipedia, Connectivity, 2020,
https://en.wikipedia.org/wiki/Connectivity_(graph_theory).

[4] M. Berg, M. Kreveld, M. Overmars, O. Schwarzkopf, *Geometria obliczeniowa. Algorytmy i zastosowania*, WNT, Warszawa 2007.

[5] Wikipedia, Ściana (teoria grafów), 2020,
https://pl.wikipedia.org/wiki/Ściana_(teoria_grafów).

[6] Andrzej Kapanowski, graphs-dict, GitHub repository, 2020,
https://github.com/ufkapano/graphs-dict/.

[7] Python Programming Language - Official Website,
https://www.python.org/.

[8] Wikipedia, Doubly connected edge list, 2020,
https://en.wikipedia.org/wiki/Doubly_connected_edge_list.

[9] Wikipedia, List comprehension, 2020,
https://en.wikipedia.org/wiki/List_comprehension

[10] Wikipedia, Breadth first search, 2020,
https://en.wikipedia.org/wiki/Breadth-first_search

[11] Wikipedia, Tree (data structure), Terminology, 2020,
https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology

41:7615550480