Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Mateusz Malczewski

Nr albumu: 1153671

Algorytmy wielokątów monotonicznych

Praca magisterska na kierunku Informatyka gier komputerowych

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

Kraków 2024

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 bardzo podziękować Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za poświęcony czas, cierpliwość i bezgraniczną pomoc dzięki której powstała ta praca.

Streszczenie

Niniejsza praca poświęcona jest implementacji i analizie wybranych algorytmów wielokątów monotonicznych. Wielokąt monotoniczny to wielokąt, który dla pewnej prostej L jest przecinany przez każdą prostą prostopadłą do L w co najwyżej dwóch punktach. Wielokąt wypukły jest monotoniczny względem dowolnej prostej L.

W pracy omówione zostały trzy algorytmy triangulacji: algorytm triangulacji wachlarzowej działający w czasie liniowym dla dowolnego wielokąta wypukłego, algorytm triangulacji wielokąta y-monotonicznego oraz algorytm triangulacji z minimalizacją długości cięciw wykorzystujący programowanie dynamiczne. Opisany został także algorytm podziału monotonicznego wielokąta prostego wykorzystujący technikę zamiatania płaszczyzny i działający w czasie liniowo-logarytmicznym. Każdy z powyższych algorytmów zaimplementowany został w wersji wykorzystującej graf jako strukturę danych wyjściowych, oraz w wersji wykorzystującej w tym celu mapę planarną. W algorytmach wykorzystujących graf złożoność obliczeniowa zgadza się z teoretyczną, a w przypadku wersji wykorzystujących mapę planarną złożoność trochę pogarsza się, co jest ceną za posiadanie informacji o strukturze topologicznej podziału. Dodatkowo algorytm triangulacji z minimalizacją długości cięciw opracowany został w dwóch wersjach: iteracyjnej i rekurencyjnej. Zaimplementowano również algorytm wyznaczania kierunków monotoniczności wielokąta działający w czasie liniowym.

Poprawność wyników wszystkich algorytmów została przetestowana, a ich złożoność obliczeniowa sprawdzona eksperymentalnie i opisana.

Słowa kluczowe: wielokąty monotoniczne, triangulacja wielokąta, podział monotoniczny wielokąta, kierunki monotoniczności

English title: Monotone polygons algorithms

Abstract

This paper is devoted to analysis and implementation of selected algorithms for monotone polygons. A monotone polygon is a polygon that, for a line L, is intersected by any line perpendicular to L in at most two points. A convex polygon is monotone with respect to any straight line L.

In this paper three triangulation algorithms are presented: a fan triangulation algorithm that works in linear time for any convex polygon, an y-monotone polygon triangulation algorithm and a triangulation algorithm which minimizes chords lengths and uses dynamic programming. A monotone polygon partitioning algorithm, which uses sweep line technique and runs in linearithmic time is also presented. Each of the above algorithms has been implemented in a version which uses a graph as an output data structure, as well as in a version using planar map for this purpose. In implementations using graphs, algorithm time complexity corresponds to theoretical, but in implementations using planar maps, time complexity is a bit worse, which is a cost for having direct access to the topological structure of a partition. Additionally, the triangulation algorithm which minimizes chords lengths was developed in two versions: iterative and recursive. An algorithm for finding monotone directions of a given polygon, which runs in linear time, is also presented.

Correctness of all algorithms was tested as well as their real computational complexity.

Keywords: monotone polygons, polygon triangulation, monotone partitioning, monotone directions

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy są wielokąty monotoniczne [1], które są uogólnieniem wielokątów wypukłych, a z drugiej strony są łatwiejsze do analizy niż ogólne wielokąty proste. Celem pracy jest przedstawienie wybranych algorytmów dla wielokątów monotonicznych oraz ich implementacja w języku Python [2]. W implementacji będzie wykorzystany pakiet *planegeometry* rozwijany na Wydziale FAIS UJ w Krakowie [3]. Przygotowane implementacje algorytmów dla wielokątów monotonicznych wejdą w skład pakietu *planegeometry*.

1.1. Problem podziału wielokąta

Jednym z ważnych problemów w geometrii obliczeniowej jest podział wielokąta (ang. *polygon partition*) [4]. Czasem używany jest termin dekompozycja wielokąta (ang. *polygon decomposition*). Jest to podział wielokąta prostego na podstawowe jednostki, które nie nakładają się na siebie, a ich suma daje początkowy wielokąt. Problem podziału wielokąta polega na znalezieniu podziału w pewnym sensie minimalnego. Przykładowo przy podziale na wielokąty wypukłe minimalizuje się liczbę wielokątów wypukłych. Podział wielokąta ma wiele zastosowań, np. przy rozpoznawaniu wzorców, kompresji danych, przetwarzaniu obrazów, itp. W wielu sytuacjach lepiej jest operować prostszymi obiektami niż ogólne wielokąty proste.

W problemie podziału wielokąta należy zdecydować, czy dopuszczalne jest dodawanie nowych punktów (punkty Steinera), czy należy wykorzystywać tylko pierwotnie zadane punkty wielokąta (dodajemy tylko cięciwy). W tej pracy skupimy się na problemach bez dodawania nowych punktów.

Dalej należy rozróżnić wielokąty z dziurami i bez dziur. Jeżeli występują dziury, to brzeg wielokąta składa się z więcej niż jednego ciągu krawędzi. Jest to znaczne utrudnienie, które czasem prowadzi do problemów NP-trudnych. W naszej pracy zawężamy się do wielokątów bez dziur.

Następna sprawa to wybór podstawowych jednostek podziału wielokąta. Najlepiej zbadany jest problem triangulacji wielokąta, czyli podziału wielokąta prostego na trójkąty [5]. Jeżeli wielokąt ma n wierzchołków, to po triangulacji otrzymamy $n-2$ trójkątów, a więc dodane zostanie $n-3$ cięciw. Algorytm triangulacji wielokąta prostego w czasie $O(n)$ podał Chazelle w roku 1991, ale ten algorytm jest uważany za bardzo skomplikowany i chyba nie udało się go zaimplementować. Z drugiej strony, w pewnych zastosowaniach podział wielokąta na trójkąty nie jest wskazany ze względu na dużą ich liczbę. Stąd rozważa się podział wielokąta na wielokąty wypukłe, wielokąty monotoniczne, wielokąty gwiaździste, czy wielokąty spiralne.

1.2. Implementacja wielokątów

Jeżeli chcemy zaimplementować wybrany algorytm podziału wielokąta, to pojawia się problem wyboru struktur danych do reprezentowania wielokąta i jego podziału. W pracy licencjackiej Gabrieli Mazur [6] opisano triangulację wachlarzową wielokąta wypukłego i triangulację wielokąta monotonicznego. Wielokąty były reprezentowane przez listy punktów (klasa Point), natomiast w wyniku triangulacji otrzymywano zbiór trójkątów (klasa Triangle), które przechowywano w kolekcji trójkątów (klasa TriangleCollection).

W naszej pracy będziemy rozważać ogólniejsze podziały, więc zdecydowaliśmy o wykorzystaniu map planarnych do reprezentowania wielokąta i jego podziału. Mapy planarne (klasa PlanarMap) opisują faktycznie grafy płaskie spójne i są one dostępne w pakiecie planegeometry. Wierzchołki wielokąta (grafu) to instancje klasy Point, krawędzie wielokąta to instancje klasy Segment. Spójne obszary na płaszczyźnie to ściany grafu i w jednolity sposób reprezentowane są elementy podziału wielokąta (trójkąty, wielokąty wypukłe, wielokąty monotoniczne) i obszar zewnętrzny wielokąta (ściana zewnętrzna grafu). W strukturze mapy planarnej każdy spójny obszar (ściana) ma automatycznie przyporządkowaną etykietę, za pomocą której można szybko znaleźć krawędzie ograniczające daną ścianę, ściany sąsiadujące z daną krawędzią, itp. Mapy planarne zostały wprowadzone w pracy licencjackiej Anny Sarnavskiej, gdzie opisano złożoną operację nakładania się map [7].

Praktyczne testy wydajnościowe implementacji wykorzystystujących mapy planarne wykazały spowolnienie wynikające z operacji wstawiania nowej krawędzi pomiędzy krawędzie wychodzące z danego wierzchołka. Z tego powodu przygotowano dodatkowe implementacje wykorzystujące strukturę danych grafu abstrakcyjnego. W ten sposób otrzymano optymalną złożoność obliczeniową, ale informacja o topologii podziału wielokąta nie jest dostępna wprost.

1.3. Struktura pracy

Praca została podzielona w następujący sposób. Rozdział 1 wprowadza do tematyki pracy, przybliża problem podziału wielokąta oraz w skrócie omawia sposób implementacji wielokątów i ich podziałów. W rozdziale 2 przedstawione zostały istotne dla pracy definicje pojęć z dziedziny geometrii obliczeniowej. Rozdział 3 opisuje używane w pracy struktury danych. Rozdział 4 poświęcony został szczegółowym opisom zaimplementowanych algorytmów. Zawiera on również listingi z kodami źródłowymi omawianych algorytmów oraz rysunki przedstawiające efekty ich działania. W rozdziale 5 przedstawione zostało podsumowanie pracy. W dodatku A umieszczone zostały testy zaimplementowanych algorytmów potwierdzające ich rzeczywistą złożoność obliczeniową.

2. Geometria obliczeniowa

W tym rozdziale przedstawione zostaną definicje pojęć z dziedziny geometrii obliczeniowej używane w tej pracy.

2.1. Wielokąty

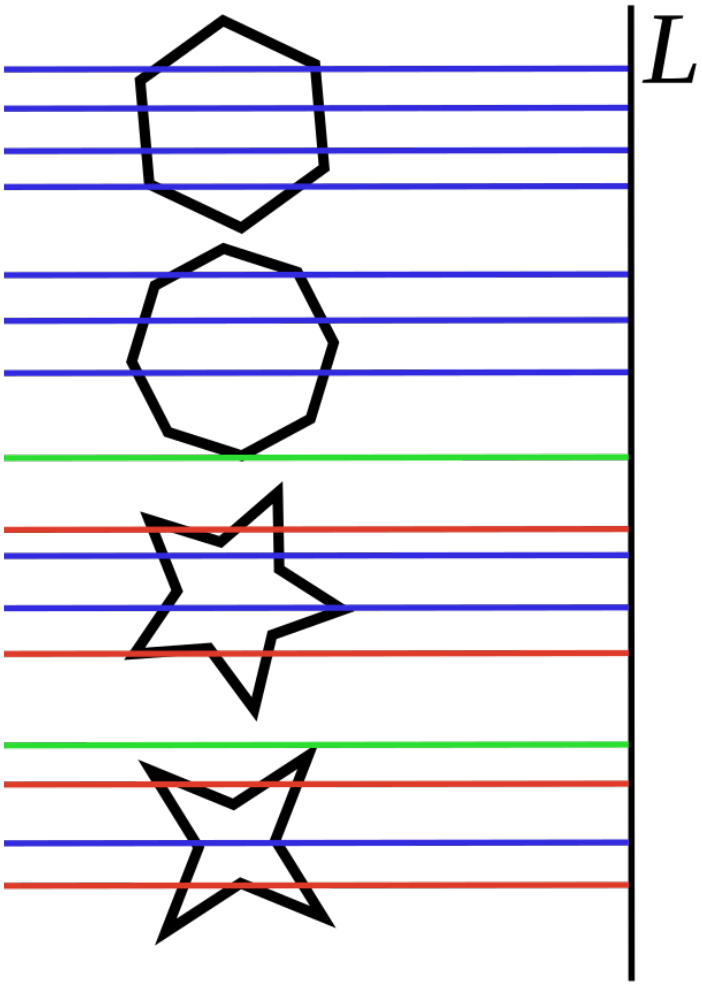
Definicja - wielokąt [8]: Wielokąt (ang. *polygon*) to dwuwymiarowa figura geometryczna tworzona przez połączone ze sobą odcinki (krawędzie) tworzące zamkniętą linię łamaną. Punkty, w których jeden odcinek kończy się, a drugi zaczyna, nazywamy wierzchołkami wielokąta. Wielokąt z n wierzchołkami posiada n krawędzi.

Definicja - wielokąt prosty [9]: Wielokąt prosty (ang. *simple polygon*) to wielokąt, którego krawędzie nie przecinają się oraz w którym nie ma dziur. Suma kątów zewnętrznych wielokąta prostego wynosi 2π, a suma kątów wewnętrznych wynosi $\pi(n-2)$. Wielokąty proste posiadają wiele zastosowań w geometrii obliczeniowej, między innymi pozwalają na szybkie sprawdzenie, czy dany punkt znajduje się wewnątrz wielokąta lub na znalezienie otoczki wypukłej w czasie liniowym, a więc szybciej niż w przypadku algorytmów działających na zbiorze punktów nie tworzącym wielokąta.

Definicja - wielokąt wypukły [10]: Wielokąt wypukły (ang. *convex polygon*) to wielokąt prosty, w którym dowolnie wybrane dwa punkty znajdujące się wewnątrz tego wielokąta możemy połączyć odcinkiem w taki sposób, że całość tego odcinka również znajduje się wewnątrz wielokąta. W wielokącie wypukłym wszystkie kąty wewnętrzne są równe lub mniejsze od π. Wielokąt wypukły, w którym wszystkie kąty wewnętrzne są mniejsze od π, nazywamy wielokątem ściśle wypukłym. Wielokąt wypukły jest otoczką wypukłą swoich wierzchołków oraz można go poddać triangulacji wachlarzowej w czasie liniowym.

Definicja - wielokąt wklęsły [11]: Wielokąt wklęsły (ang. *concave polygon*) to wielokąt prosty, który posiada co najmniej jeden kąt wewnętrzny większy niż π. Innymi słowy, wielokąt wklęsły to taki wielokąt prosty, który nie jest wypukły, a więc taki, w którym istnieją pary punktów leżący wewnątrz wielokąta, których odcinek łączący nie znajduje się w całości wewnątrz wielokąta. Wielokąt wklęsły można podzielić na zbiór wielokątów wypukłych w czasie wielomianowym.

Definicja - wielokąt monotoniczny [1]: Wielokąt monotoniczny (ang. *monotone polygon*) to wielokąt, który dla pewnej prostej L jest przecinany przez każdą prostą prostopadłą do L w co najwyżej dwóch punktach. Dobrze obrazuje to rysunek 2.1. Definicję można również rozszerzyć, aby uwzględniała wielokąty, które posiadają krawędzie prostopadłe do prostej L. Wówczas wielokąty posiadające takie krawędzie nazywane są słabo monotonicznymi, a wielokąty ich nie posiadające nazywane są silnie monotonicznymi. Wielokąty wypukłe są monotoniczne względem każdej prostej.



Rysunek 2.1. Wizualizacja monotoniczności przykładowych wielokątów względem prostej L. Niebieskie linie obrazują proste przecinające wielokąt w dwóch punktach, zielone linie pokazują proste przecinające wielokąt w jednym punkcie, a czerwone linie obrazują proste przecinające wielokąt w więcej niż dwóch punktach. Dwa pierwsze wielokąty są monotoniczne względem prostej L natomiast dwa ostatnie nie są monotoniczne względem tej prostej [1].

2.2. Podział wielokąta

Podział wielokąta (ang. *polygon partition*) to zbiór wielokątów prostszych od początkowego wielokąta, które nie przecinają się, a których suma jest równa początkowemu wielokątowi. Najczęściej wykorzystywane w tym celu są trójkąty (triangulacja), ale mogą to być także bardziej ogólne wielokąty, na przykład wielokąty monotoniczne (podział monotoniczny) [4].

Częstym zastosowaniem podziału wielokąta jest redukcja złożonego kształtu do zbioru prostszych, co ułatwia czy też w ogóle umożliwia późniejsze przetworzenie wielokąta przez dany algorytm.

2.3. Triangulacja wielokąta

Triangulacja wielokąta (ang. *polygon triangulation*) to podział wielokąta prostego na zbiór nieprzecinających się trójkątów, które razem tworzą ten sam obszar, co oryginalny wielokąt. Polega ona na dodaniu do wielokąta dodatkowych przekątnych pomiędzy wierzchołkami w taki sposób, aby każde dwa utworzone trójkąty miały ze sobą wspólny wierzchołek lub krawędź [5].

Istnieje wiele algorytmów triangulacji wielokątów, na przykład każdy wielokąt wypukły lub wklęsły z jednym wierzchołkiem wklęsłym może zostać poddany triangulacji wachlarzowej, czyli triangulacji, w której dodajemy krawędzie z jednego wierzchołka do wszystkich pozostałych.

3. Implementacja

W rozdziale tym opisane zostały podstawowe obiekty i struktury danych wchodzące w skład pakietu *planegeometry* wykorzystywane w algorytmach opisanych w tej pracy.

3.1. Figury geometryczne

Pakiet *planegeometry* zawiera w sobie implementację wielu podstawowych struktur geometrycznych. W tej sekcji opisane zostały te z nich, które zostały użyte w zaimplementowanych algorytmach.

3.1.1. Punkt

Punkt reprezentowany jest przez klasę Point. Przechowuje wartości współrzędnych x i y określających położenie w układzie kartezjańskim. Klasa Point przeciąża operatory podstawowych działań matematycznych, takich jak dodawanie, odejmowanie czy mnożenie, dzięki czemu możemy wykonywać operacje na jej instancjach w taki sam sposób jak na typach wbudowanych.

3.1.2. Odcinek

Odcinek reprezentowany jest przez klasę Segment. Przechowuje parę punktów (instancje klasy Point), z których jeden punkt jest punktem początkowym, a drugi końcowym. Dodatkowo klasa Segment implementuje wiele metod pomocniczych pozwalających między innymi na obliczenie długości odcinka, sprawdzenie równoległości i prostopadłości odcinków, wyznaczenie punktu przecięcia dwóch odcinków. Interfejs klasy Segment jest zgodny z interfejsem klasy Edge, dzięki czemu odcinki można traktować jak krawędzie w grafie, gdzie wierzchołkami grafu są instancje klasy Point.

3.1.3. Wielokąt

Wielokąt reprezentowany jest przez klasę Polygon. Przechowuje listę punktów (instancje klasy Point), których liczba musi być większa od dwóch, bo w przeciwnym wypadku nie jest to wielokąt, więc program rzuci wyjątek. Podobnie jak w przypadku klasy definiującej odcinek, klasa Polygon posiada wiele metod pomocniczych operujących na wielokącie, które pozwalają na przykład na sprawdzenie orientacji wielokąta, obliczenie prostokąta ograniczającego wielokąt, a także na sprawdzenie, czy wielokąt jest wielokątem prostym lub wypukłym. Instancja tej klasy przyjmowana jest jako argument wejściowy przez każdy opisany w tej pracy algorytm.

3.2. Struktury danych

W pakiecie *planegeometry* znajduje się także wiele struktur danych umożliwiających implementację algorytmów działających w optymalnym czasie. Opisane poniżej zostały te, które zastosowanie znalazły w algorytmach zaimplementowanych w niniejszej pracy.

3.2.1. Mapa planarna

Struktura danych reprezentowana przez klasę PlanarMap. Przechowuje dane w formie płaskiego grafu spójnego. Wierzchołki grafu to instancje klasy Point, a krawędzie to instancje klasy Segment. Spójne obszary na płaszczyźnie to ściany grafu, a elementy podziału wielokąta (trójkąty, wielokąty wypukłe, wielokąty monotoniczne) i obszar zewnętrzny wielokąta (ściana zewnętrzna grafu) reprezentowane są w sposób jednolity. Implementacja tej struktury bazuje na podwójnie łączonej liście krawędzi [12]. Mapa planarna składa się z czterech słowników, przechowujących odpowiednio informacje o następnej krawędzi wychodzącej z danego wierzchołka, poprzedniej krawędzi wychodzącej z danego wierzchołka, jednej z krawędzi obiegających daną ścianę oraz o ścianie obieganej przez daną krawędź. Operując w odpowiedni sposób na tych słownikach, jesteśmy w stanie odtworzyć strukturę topologiczną wielokąta lub jego podziału. Klasa PlanarMap zawiera wiele metod pomocniczych pozwalających między innymi na dodawanie nowych wierzchołków czy krawędzi do mapy, podział jednej krawędzi na dwie czy metodę pozwalającą na obliczenie mapy powstałej w przypadku nałożenia na siebie dwóch map. Struktura mapy planarnej wykorzystana została w algorytmach triangulacji wachlarzowej, triangulacji wielokąta y-monotonicznego, podziału monotonicznego wielokąta prostego oraz w algorytmie triangulacji z minimalizacją długości cięciw.

3.2.2. Graf

Struktura danych reprezentowana przez klasę Graph. Pozwala na przechowywanie danych w formie grafu i w przypadku wielu algorytmów pozwala na uzyskanie optymalnej złożoności obliczeniowej, co nie zawsze możliwe jest w przypadku korzystania z mapy planarnej. Podobnie jak klasa PlanarMap, klasa Graph posiada w sobie metody pomocnicze pozwalające na dodawanie, sprawdzanie czy usuwanie wierzchołków i krawędzi. Struktura grafu wykorzystana została w alternatywnych wersjach algorytmów triangulacji wachlarzowej, triangulacji wielokąta y-monotonicznego, podziału monotonicznego wielokąta prostego i triangulacji z minimalizacją długości cięciw.

3.2.3. Pozioma miotła

Struktura danych dla miotły poziomej (ang. *sweep line*) reprezentowana jest przez klasę SlowTreeX. Jest to klasa symulująca zachowanie zmodyfikowanego drzewa AVL dla problemu przecinania krawędzi. Struktura ta używana jest w algorytmie podziału monotonicznego wielokąta prostego do utworzenia poziomej miotły zamiatającej płaszczyznę z góry na dół. Odcinki w miotle

są sortowane względem współrzędnej X. Istotne jest, że przy przesuwaniu miotły w dół zmieniają się klucze użyte do sortowania (współrzędna X). W niektórych algorytmach geometrii obliczeniowej wykorzystuje się miotłę pionową zamiatającą płaszczyznę z lewa na prawo, przy czym odcinki w miotle sortowane są względem współrzędnej Y. Wtedy można użyć drugiej klasy SlowTreeY z modułu slowtrees.

Pakiet *planegeometry* zawiera w sobie implementację zwykłego drzewa AVL, ale jego użycie dla tego problemu nie było możliwe. Potrzebne byłoby zbudowanie zmodyfikowanego drzewa AVL z dodatkowymi metodami, więc dla prostoty użyto klasy SlowTreeX. Pewne metody tej klasy są wolniejsze niż w drzewie AVL, ale były wystarczające dla naszych zastosowań.

4. Algorytmy

W tym rozdziale zaprezentowane zostaną wybrane algorytmy wielokątów monotonicznych razem ze szczegółowymi opisami ich działania oraz analizami złożoności obliczeniowej. Rozpoczniemy od triangulacji wachlarzowej wielokąta wypukłego i triangulacji wielokąta monotonicznego wyrażonych w języku map planarnych i grafów. Dalej pokażemy podział monotoniczny wielokąta prostego oraz wyznaczanie kierunków monotoniczności wielokąta prostego. Na końcu rozważymy problem triangulacji o najmniejszej wadze dla wielokąta wypukłego, gdzie minimalizowana jest długość cięciw dodawanych podczas triangulacji.

4.1. Triangulacja wachlarzowa wielokąta wypukłego

Przedstawimy triangulację wachlarzową wielokąta prostego wypukłego zaimplementowaną z użyciem map planarnych. Dopuszczamy występowanie wierzchołków z kątem wewnętrznym 180°. Triangulacja wachlarzowa możliwa jest jedynie dla wielokątów wypukłych lub dla wielokątów wklęsłych posiadających jeden wierzchołek wklęsły. Przypadek ten trzeba jednak opatrzyć specjalnym kodem, gdyż podstawowy algorytm triangulacji wachlarzowej nie radzi sobie z takimi wielokątami.

Dane wejściowe: Obiekt klasy Polygon opisujący wielokąt wypukły, którego triangulację wachlarzową chcemy wyznaczyć. Dopuszczamy występowanie wierzchołków z kątem wewnętrznym 180°.

Dane wyjściowe: Obiekt klasy PlanarMap przechowujący wyznaczoną triangulację wachlarzową wielokąta wypukłego.

Problem: Triangulacja wachlarzowa wielokąta uwzględniająca wierzchołki współliniowe.

Opis algorytmu: Na początku działania algorytm sprawdza, czy podany wielokąt nie jest specjalnym przypadkiem, który należy rozpatrzeć w szczególny sposób.

Najprostszym specjalnym przypadkiem jest wielokąt, który składa się jedynie z trzech wierzchołków, co oznacza, że jest on trójkątem, więc triangulacja nie jest potrzebna.

Trudniejszym do rozwiązania specjalnym przypadkiem jest wielokąt, który jest trójkątem posiadającym dodatkowe współliniowe wierzchołki, ale tylko na jednym boku. Aby wyznaczyć triangulację wachlarzową takiego wielokąta, musimy w pierwszej kolejności odnaleźć odpowiedni wierzchołek, od

którego możliwe jest rozpoczęcie właściwej części algorytmu triangulacji. Przechodzimy po liście wierzchołków, szukając takiego wierzchołka, którego kąt wynosi 180°. Jeżeli w wyniku poszukiwań odnajdziemy taki wierzchołek, w głównej części algorytmu triangulacji wierzchołki rozpatrujemy rozpoczynając od znalezionego wierzchołka. W przeciwnym wypadku, w wielokącie nie ma żadnego współliniowego wierzchołka, więc dalszą część algorytmu rozpoczynamy po prostu od początkowego wierzchołka wielokąta.

Wyjściowym krokiem głównej części algorytmu jest detekcja pierwszego zakrętu wielokąta (czyli pierwszego wierzchołka, którego kąt jest mniejszy niż 180°). Sprawdzamy orientację kolejnych trójek wierzchołków, aż do momentu odnalezienia poszukiwanego wierzchołka. Gdy znajdziemy wierzchołek o kącie mniejszy niż 180°, zapisujemy w zmiennej pomocniczej wierzchołek poprzedzający go i przerywamy pętle. Następnie szukamy drugiego zakrętu, rozpoczynając poszukiwania od wierzchołka występującego po poprzednio znalezionym zakręcie. Sprawdzamy kolejne trójki wierzchołków.

W przypadku, gdy sprawdzana trójka nie jest zakrętem, dodajemy do mapy planarnej odcinek pomiędzy zapisanym odcinkiem poprzedzającym pierwszy zakręt z obecnie rozważanym wierzchołkiem i przechodzimy do sprawdzania kolejnej trójki wierzchołków.

Jeśli obecnie testowana trójka jest zakrętem, zapisujemy w zmiennej pomocniczej wierzchołek poprzedzający drugi zakręt, a następnie dodajemy do mapy planarnej odcinek łączący wierzchołek poprzedzający pierwszy zakręt z wierzchołkiem poprzedzający drugi zakręt i przerywamy pętle.

W ostatniej części algorytmu przechodzimy po wszystkich pozostałych wierzchołkach, rozpoczynając od wierzchołka znajdującego się za drugim zakrętem, aż do momentu natrafienia na zapisany wierzchołek poprzedzający pierwszy zakręt i dla każdego wierzchołka dodajemy do mapy planarnej odcinek pomiędzy tym wierzchołkiem a wierzchołkiem poprzedzającym drugi zakręt. Gdy rozważymy wszystkie wierzchołki, następuje zakończenie algorytmu, a na wyjściu otrzymujemy mapę planarną przechowującą obliczoną triangulację wachlarzową wejściowego wielokąta.

Rysunek 4.1 przedstawia wynikową triangulację wachlarzową dla przykładowego wielokąta wypukłego.

Złożoność: Teoretyczna czasowa algorytmu wynosi $O(n)$, jednak tak samo jak w przypadku innych algorytmów wykorzystanie mapy planarnej powoduje jej pogorszenie. Złożoność obliczeniowa wersji wykorzystującej graf zgadza się z teoretyczną.

Uwaga: Algorytm zakłada, że orientacja wejściowego wielokąta jest przeciwna do ruchu wskazówek zegara. W przypadku orientacji zgodnej z ruchem wskazówek zegara zostanie rzucony wyjątek.

Listing 4.1. Moduł fan2.

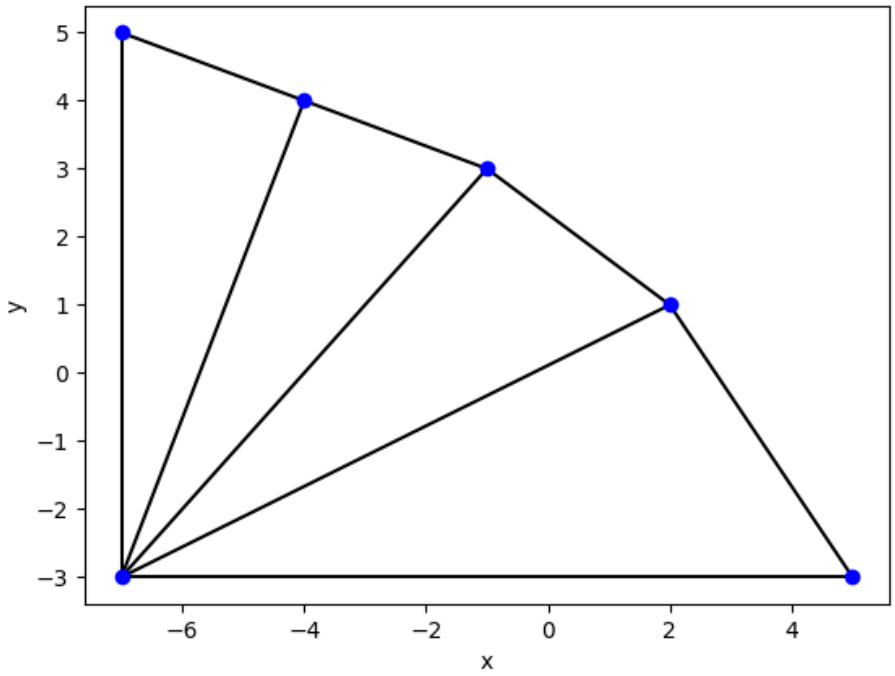
```
#!/usr/bin/env python3

try:
    range = xrange
except NameError:    # Python 3
```

```
    pass

from planegeometry.structures.points import Point
from planegeometry.structures.segments import Segment
from planegeometry.structures.polygons import Polygon
from planegeometry.structures.planarmaps import PlanarMap
from planegeometry.algorithms.geomtools import orientation

class FanTriangulationPM:
    """Fan triangulation of a simple polygon in O(n) time."""

    def __init__(self, polygon):
        if polygon.orientation(test_is_simple=False) < 0:
            raise ValueError("clockwise orientation detected")
        self.point_list = polygon.point_list    # nie ma kopiowania punktow
        self.n = len(self.point_list)
        self.planar_map = PlanarMap(polygon)

    def run(self):
        if self.n == 3:
            return

        ic = 0

        for i in range(self.n):
            if orientation(self.point_list[i],
                           self.point_list[(i+1) % self.n],
                           self.point_list[(i+2) % self.n]) == 0:
                ic = (i+1) % self.n   # nie ma zakretu
                break

        for i in range(self.n):
            j = (ic + i) % self.n
            if orientation(self.point_list[j],
                           self.point_list[(j+1) % self.n],
                           self.point_list[(j+2) % self.n]) > 0:
                ia = j
                break

        for i in range(self.n):
            j = (ia + 1 + i) % self.n   # zaczynamy od pierwszego zakretu
            k = (ia + 2 + i) % self.n   # za pierwszym zakretem
            if orientation(self.point_list[k],
                           self.point_list[(k+1) % self.n],
                           self.point_list[(k+2) % self.n]) > 0:
                ib = k
                segment = Segment(self.point_list[ia], self.point_list[ib])
                self.planar_map.add_chord(segment)
                break
            else:
                segment = Segment(self.point_list[ia], self.point_list[k])
                self.planar_map.add_chord(segment)

        for i in range(self.n):
            j = (ib + 1 + i) % self.n
            k = (ib + 2 + i) % self.n
            if k == ia:
```

```
                    break
                segment = Segment(self.point_list[ib], self.point_list[k])
                self.planar_map.add_chord(segment)
```

Listing 4.2. Przykład użycia algorytmu triangulacji wachlarzowej wielokąta wypukłego

```
>>> from points import Point
>>> from polygons import Polygon
>>> from fan2 import FanTriangulationPM as Triangulation
>>> point_list = [Point(-7,-3), Point(5,-3), Point(2,1), Point(-1,3),
    Point(-4,4), Point(-7,5)]
>>> polygon = Polygon(*point_list)
>>> algorithm = Triangulation(polygon)
>>> algorithm.run()
>>> algorithm.planar_map.show()
```



Rysunek 4.1. Przykładowa triangulacja wachlarzowa wielokąta wypukłego.

4.2. Triangulacja wielokąta y-monotonicznego

Dla wielokąta y-monotonicznego przedstawimy jego triangulację zapisaną w języku map planarnych. Do wyznaczenia triangulacji możemy wykorzystać algorytm zachłanny, polegający na przejściu po wierzchołkach należących do łańcuchów tego wielokąta z góry na dół, dodając przekątne, gdy jest to możliwe.

Dane wejściowe: Obiekt klasy Polygon opisujący wielokąt y-monotoniczny, którego triangulację chcemy wyznaczyć.

Dane wyjściowe: Obiekt klasy PlanarMap przechowujący wyznaczoną triangulację wielokąta.

Problem: Triangulacja wielokąta monotonicznego.

Opis algorytmu: Algorytm rozpoczynamy od utworzenia mapy planarnej (obiektu klasy PlanarMap) odpowiadającej wielokątowi, dla którego obliczana będzie triangulacja. Następnie przechodzimy po wierzchołkach wielokąta w celu odnalezienia dwóch końcowych wierzchołków względem osi Y, czyli wierzchołka z maksymalną wartością y i wierzchołka z minimalną wartością y. Znając te wierzchołki, jesteśmy w stanie zbudować dwa łańcuchy, lewy i prawy, których znajomość pozwala na wyznaczenie triangulacji. W implementacji oba te łańcuchy przechowywane są w ramach jednej listy, zawierającej wszystkie wierzchołki wielokąta, razem z informacją, do jakiego łańcucha należą, posortowane malejąco względem wartości ich współrzędnej y.

Aby osiągnąć optymalną złożoność obliczeniową algorytmu, potrzebujemy pomocniczej struktury danych, która przechowywać będzie aktualnie rozważane wierzchołki. Strukturą tą jest stos, ponieważ wystarczy nam możliwość dodawania elementu na stos oraz ściąganie górnego elementu, czyli operacji, które w przypadku stosu wykonują się w czasie $O(1)$.

Po podzieleniu wierzchołków na dwa łańcuchy rozpoczynamy główną część algorytmu odpowiadającą za wyznaczanie triangulacji. Dodajemy na stos dwa pierwsze wierzchołki z posortowanej listy, a następnie przechodzimy po wszystkich pozostałych wierzchołkach. Aktualnie rozważany wierzchołek porównujemy z wierzchołkiem znajdującym się obecnie na szczycie stosu. Występuje wówczas jedna z dwóch sytuacji: aktualnie rozważany wierzchołek może należeć do tego samego łańcucha co wierzchołek na szczycie stosu lub mogą one należeć do różnych łańcuchów.

W przypadku, gdy wierzchołki należą do różnych łańcuchów, postępujemy w następujący sposób: dopóki na stosie znajduje się więcej niż jeden element, ściągamy ze stosu pierwszy element i dodajemy do obecnej mapy planarnej odcinek pomiędzy aktualnie rozważanym wierzchołkiem a wierzchołkiem ściągniętym ze stosu. W momencie, gdy na stosie zostanie tylko jeden wierzchołek, ściągamy go ze stosu, gdyż nie jest on już potrzebny. Na koniec dodajemy na stos wierzchołki, między którymi dodaliśmy ostatni odcinek.

W sytuacji, gdy wierzchołki należą do tego samego łańcucha, sposób postępowania wygląda inaczej. Ściągamy ze stosu pierwszy element i zapisujemy go w zmiennej pomocniczej. Następnie sprawdzamy, czy orientacja trójki wierzchołków złożonej z aktualnie rozważanego wierzchołka, wierzchołka ściągniętego ze stosu oraz wierzchołka obecnie znajdującego się na szczycie stosu odpowiada orientacji łańcucha, do którego należy aktualnie rozważany wierzchołek.

Jeżeli orientacja jest taka sama, dodajemy do mapy planarnej odcinek pomiędzy aktualnie rozważanym wierzchołkiem a wierzchołkiem na szczycie stosu, po czym ściągamy ten element ze stosu.

Finalnie, zarówno w przypadku gdy orientacja trójki wierzchołków była równa orientacji łańcucha aktualnego wierzchołka, jak i w przeciwnym wypadku dodajemy na stos ostatni ściągnięty wierzchołek oraz wierzchołek aktualnie rozważany.

Aby zakończyć algorytm, musimy jeszcze w sposób szczególny rozważyć ostatni wierzchołek, będący zakończeniem obu łańcuchów. W momencie rozpatrywania ostatniego wierzchołka ściągamy ze stosu pierwszy element, a następnie, dopóki ilość elementów na stosie jest większa niż jeden, zdejmujemy wierzchołek ze szczytu stosu i dodajemy do mapy planarnej odcinek pomiędzy nim a ostatnim wierzchołkiem na liście wierzchołków. Po przetworzeniu wszystkich elementów na stosie następuje zakończenie algorytmu, a na wyjściu otrzymujemy mapę planarną przechowującą obliczoną triangulację wejściowego wielokąta.

Rysunek 4.2 przedstawia wynikową triangulację dla przykładowego wielokąta y-monotonicznego.

Złożoność: Złożoność czasowa algorytmu w teorii wynosi $O(n)$. W praktyce są dwa miejsca, w których mamy pogorszenie złożoności. Pierwsze miejsce to łączenie dwóch łańcuchów wierzchołków, lewego i prawego, w jeden łańcuch. Dla prostoty użyto złączenia łańcuchów w jeden łańcuch, a następnie wykonano sortowanie (timsort) w czasie $O(n \log n)$. Można to poprawić wykonując złączenie (*merge*) dwóch łańcuchów posortowanych w jeden łańcuch posortowany, ale w praktyce domyślne sortowanie Pythona jest wystarczająco szybkie.

Drugie miejsce pogarszające złożoność to wstawianie cięciwy do wielokąta metodą planar_map.add_chord(segment). Czas pracy metody zależy od stopni końcowych wierzchołków, ponieważ należy znaleźć właściwe miejsce cięciwy pomiędzy istniejącymi krawędziami wychodzącymi z wierzchołków. Przy obu końcach cięciwy wykonywane jest sortowanie krawędzi wychodzących względem kąta, a następnie znajduje się krawędzie sąsiadujące z obu stron z dodawaną cięciwą. Nie jest jasne, czy można przyspieszyć te czynności.

Testy porównujące wydajność dwóch różnych implementacji triangulacji wielokąta monotonicznego pokazują, że implementacja z mapami planarnymi jest dużo wolniejsza od implementacji wykorzystującej graf. Wniosek praktyczny jest taki, że jeżeli nie potrzebujemy informacji o strukturze topologicznej triangulacji, to lepiej użyć implementacji z grafem. Testy triangulacji opisano w dodatku A.

Uwaga: Podczas sortowania wierzchołków po wartości współrzędnych y może wystąpić przypadek, gdy współrzędna y wierzchołków będzie równa. W takim wypadku stosowane jest drugie kryterium sortowania, czyli sortowanie rosnące względem wartości x.

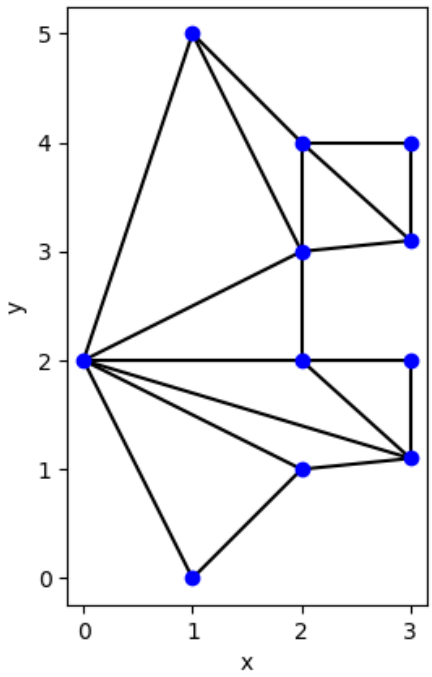
Listing 4.3. Moduł triangulation2.

```
#!/usr/bin/env python3

try:
    range = xrange
except NameError:    # Python 3
    pass

from planegeometry.structures.points import Point
from planegeometry.structures.segments import Segment
from planegeometry.structures.polygons import Polygon
from planegeometry.structures.planarmaps import PlanarMap
from planegeometry.algorithms.geomtools import orientation

class Chain:
    LEFT = 0
    RIGHT = 1

class YMonotoneTriangulationPM:
    """Triangulation of a strictly y-monotone polygon in O(n) time."""

    def __init__(self, polygon):
        if polygon.orientation(test_is_simple=False) < 0:
            raise ValueError("clockwise orientation detected")
        self.point_list = polygon.point_list
        self.n = len(self.point_list)
        self.planar_map = PlanarMap(polygon)
        self.ulist = []    # for pairs (point, chain)
        self.stack = []    # for pairs (point, chain)

    def run(self):
        self.merge_chains()
        self.stack.append(self.ulist[0])
        self.stack.append(self.ulist[1])
        for i in range(2, self.n-1):
            point1, chain1 = self.ulist[i]
            point2, chain2 = self.stack[-1]
            if chain1 != chain2:
                while len(self.stack) > 1:
                    point2, chain2 = self.stack.pop()
                    segment = Segment(point1, point2)
                    self.planar_map.add_chord(segment)

                self.stack.pop()
                self.stack.append(self.ulist[i-1])
                self.stack.append(self.ulist[i])
            else:
                point2, chain2 = self.stack.pop()
                point3, chain3 = self.stack[-1]
                if chain1 == Chain.LEFT:
                    orient = -1
                else:
                    orient = +1
                while orientation(point1, point2, point3) == orient:
                    segment = Segment(point1, point3)
                    self.planar_map.add_chord(segment)
                    point2, chain2 = self.stack.pop()
```

```
                    if len(self.stack) == 0:
                        break
                    else:
                        point3, chain3 = self.stack[-1]
                self.stack.append((point2, chain2))
                self.stack.append(self.ulist[i])

        point1, chain1 = self.ulist[self.n-1]
        point2, chain2 = self.stack.pop()
        while len(self.stack) > 1:
            point3, chain3 = self.stack.pop()
            segment = Segment(point1, point3)
            self.planar_map.add_chord(segment)
            point2 = point3

    def merge_chains(self):
        """Preparing self.ulist with sorted pairs (point, chain)."""
        i_max = max(range(self.n), key=lambda i: (self.point_list[i].y))
        i_min = min(range(self.n), key=lambda i: (self.point_list[i].y))
        left_chain = []
        right_chain = []
        for i in range(self.n):
            j = (i_max + i) % self.n
            if j == i_min:
                break
            left_chain.append(self.point_list[j])
        for i in range(self.n):
            j = (i_min + i) % self.n
            if j == i_max:
                break
            right_chain.append(self.point_list[j])
        right_chain.reverse()

        self.ulist.extend((point, Chain.LEFT) for point in left_chain)
        self.ulist.extend((point, Chain.RIGHT) for point in right_chain)

        self.ulist.sort(key=lambda item: (item[0].y, -item[0].x), reverse=True)
```

Listing 4.4. Przykład użycia algorytmu triangulacji wielokąta y-monotonicznego

```
>>> from points import Point
>>> from polygons import Polygon
>>> from triangulation2 import YMonotoneTriangulationPM as Triangulation
>>> point_list = [Point(1, 0), Point(2, 1), Point(3, 1.1),
    Point(3, 2), Point(2, 2), Point(2, 3), Point(3, 3.1),
    Point(3, 4), Point(2, 4), Point(1, 5), Point(0, 2)]
>>> polygon = Polygon(*point_list)
>>> algorithm = Triangulation(polygon)
>>> algorithm.run()
>>> algorithm.planar_map.show()
```



Rysunek 4.2. Przykładowa triangulacja wielokąta y-monotonicznego.

4.3. Podział monotoniczny wielokąta prostego

Dla dowolnego wielokąta prostego przedstawimy jego podział monotoniczny zapisany w języku map planarnych. Aby wyznaczyć podział zastosowaliśmy algorytm wykorzystujący technikę zamiatania płaszczyzny (ang. *sweep line algorithm*), który przechodzi po wierzchołkach wielokąta od góry do dołu i rozpatruje je w różny sposób w zależności od typu wierzchołka. W przypadku naszej implementacji wynikowe wielokąty będą monotoniczne względem osi Y. Implementacja bazuje na opisie z książki de Berga [13].

Dane wejściowe: Obiekt klasy Polygon opisujący wielokąt prosty, którego podział monotoniczny chcemy wyznaczyć.

Dane wyjściowe: Obiekt klasy PlanarMap przechowujący wyznaczony podział monotoniczny wielokąta.

Problem: Podział monotoniczny wielokąta prostego.

Opis algorytmu: Głównym celem algorytmu dzielącego wielokąt prosty na wielokąty monotoniczne jest pozbycie się wszystkich wierzchołków zwrotnych (ang. *turn vertex*) wielokąta. Wierzchołek zwrotny to taki wierzchołek, w którym zmieniamy kierunek poruszania, podczas przechodzenia po kolei przez wierzchołki łańcucha wielokąta. Wierzchołka takiego pozbywamy

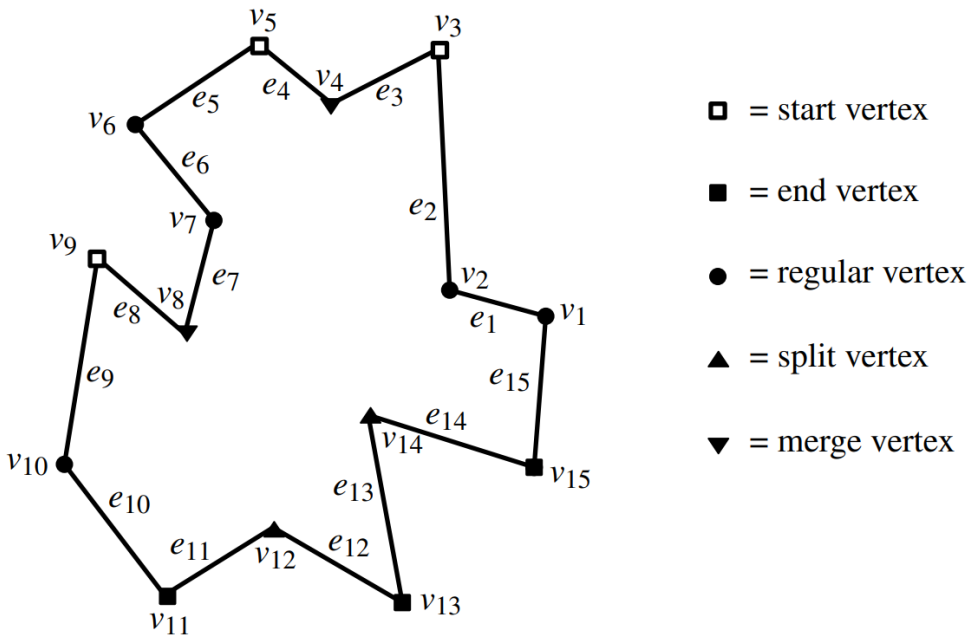
się poprzez dodanie odcinka wychodzącego z tego wierzchołka, łączącego go z innym wierzchołkiem, wybieranym w różny sposób w zależności od typu wierzchołka. Odcinek ten dzieli dotychczasowy wielokąt na dwa wielokąty, a wierzchołek zwrotny, z którego poprowadziliśmy odcinek, staje się regularnym wierzchołkiem w obu powstałych wielokątach.

W algorytmie łącznie rozróżniamy pięć rodzajów wierzchołków, z których każdy jest rozpatrywany w inny sposób. Cztery typy wierzchołków opisują różnego rodzaju wierzchołki zwrotne i są to: wierzchołki startowe (ang. *start vertices*), wierzchołki końcowe (ang. *end vertices*), wierzchołki dzielące (ang. *split vertices*) oraz wierzchołki scalające (ang. *merge vertices*). Piątym typem wierzchołków rozróżnianym przez algorytm są wierzchołki regularne (ang. *regular vertices*) i są to wszystkie pozostałe wierzchołki, które nie są wierzchołkami zwrotnymi. Typy wierzchołków zdefiniowane są następująco (rys. 4.3):

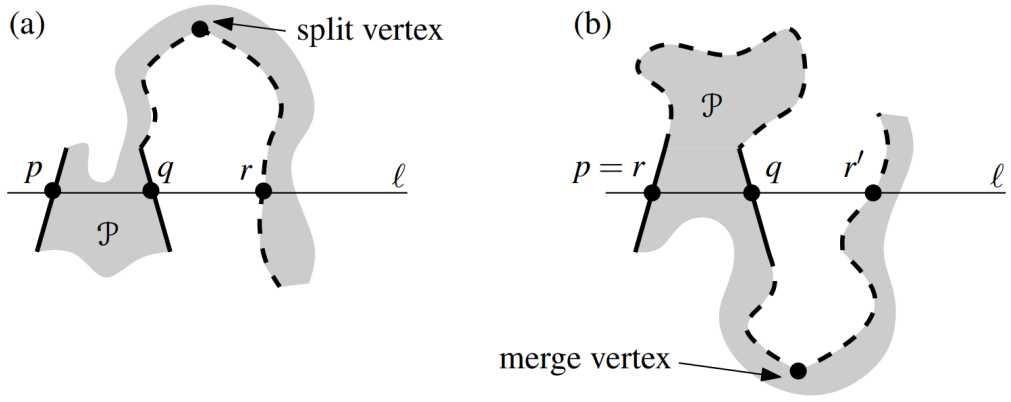
— Wierzchołek jest wierzchołkiem startowym, gdy sąsiadujące z nim wierzchołki mają mniejszą współrzędną y niż on sam, oraz gdy kąt wewnętrzny między nimi wynosi mniej niż 180°.
— Wierzchołek jest wierzchołkiem końcowym, gdy sąsiadujące z nim wierzchołki mają większą współrzędną y niż on sam, oraz gdy kąt wewnętrzny między nimi wynosi mniej niż 180°.
— Wierzchołek jest wierzchołkiem dzielącym, gdy sąsiadujące z nim wierzchołki mają mniejszą współrzędną y niż on sam, oraz gdy kąt wewnętrzny między nimi wynosi więcej niż 180°.
— Wierzchołek jest wierzchołkiem scalającym, gdy sąsiadujące z nim wierzchołki mają większą współrzędną y niż on sam, oraz gdy kąt wewnętrzny między nimi wynosi więcej niż 180°.
— Wierzchołek jest wierzchołkiem regularnym, gdy jeden z sąsiadujących z nim wierzchołków ma mniejszą współrzędną y niż on sam, a drugi sąsiadujący wierzchołek większą.

Źródłem niemonotoniczności danego wielokąta są wierzchołki dzielące i wierzchołki scalające. Inaczej mówiąc, dany wielokąt nie jest y-monotoniczny, jeżeli posiada wierzchołek dzielący albo wierzchołek scalający. Obrazuje to rysunek 4.4.

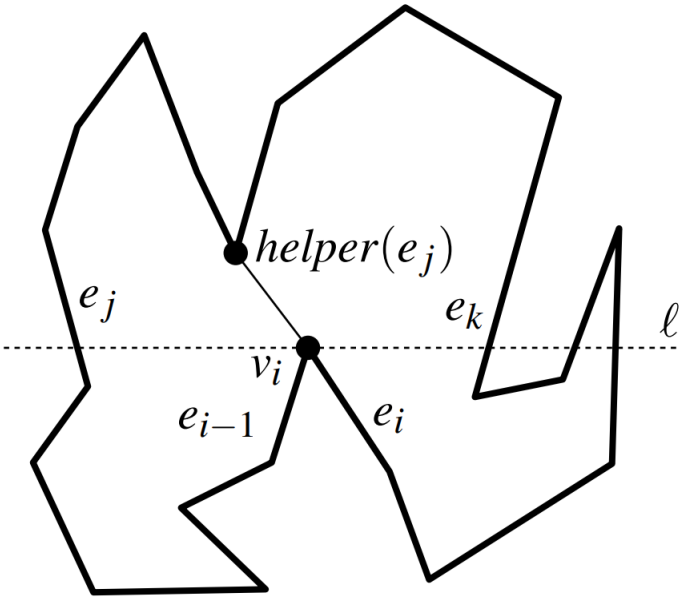
Wynika z tego, że pozbywając się wierzchołków dzielących i scalających, otrzymujemy wielokąt y-monotoniczny, więc dzieląc dany wielokąta na wielokąty nieposiadające tych wierzchołków, otrzymujemy podział wielokąta na wielokąty monotoniczne. Aby pozbyć się wierzchołka dzielącego, musimy połączyć go odcinkiem z wierzchołkiem leżącym nad nim. Gdy podczas zamiatania płaszczyzny miotła natrafi na wierzchołek dzielący, próbujemy połączyć go z najniżej położonym wierzchołkiem znajdującym się nad rozpatrywanym wierzchołkiem. Aby w prosty sposób wyznaczyć taki wierzchołek, podczas zamiatania płaszczyzny zapisujemy do zmiennej pomocniczej informację o obecnie najniżej położonym wierzchołku znajdującym się nad linią płaszczyzny, dla krawędzi wielokąta na lewo od ówcześnie rozpatrywanego wierzchołka. Sprawdzając tą informację dla krawędzi na lewo od obecnie rozpatrywanego wierzchołka dzielącego wiemy z jakim wierzchołkiem należy go połączyć odcinkiem. Przykład dodawania odcinka wychodzącego z wierzchołka dzielącego można zobaczyć na rysunku 4.5.



Rysunek 4.3. Typy wierzchołków rozpatrywane przez algorytm [13].



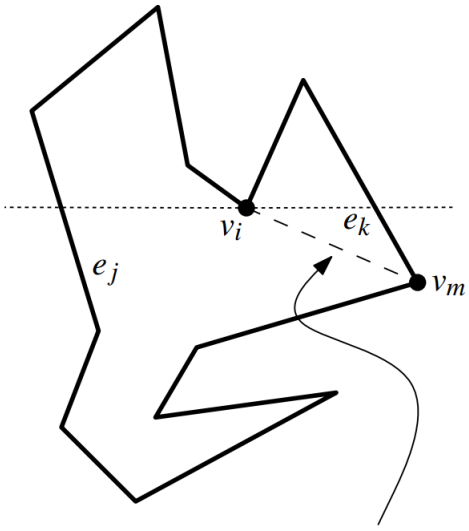
Rysunek 4.4. Dowód niemonotoniczności wielokąta w przypadku występowania wierzchołka dzielącego lub scalającego [13].



Rysunek 4.5. Przykład dodawania odcinka wychodzącego z wierzchołka dzielącego [13].

Sposób postępowania w przypadku wierzchołka scalającego jest trochę bardziej skomplikowany, ponieważ w przeciwieństwie do wierzchołka dzielącego, musimy połączyć go odcinkiem z wierzchołkiem znajdującym się pod nim, a to oznacza, że będzie on połączony z wierzchołkiem, do którego nie dotarła jeszcze miotła zamiatająca płaszczyznę. Możemy jednak postąpić w podobny sposób. Gdy miotła natrafi na wierzchołek scalający, zapisujemy go w zmiennej pomocniczej dla krawędzi wielokąta na lewo od rozpatrywanego wierzchołka, tak samo jak w przypadku wierzchołków innego typu. Chcemy połączyć ten wierzchołek z wierzchołkiem położonym najwyżej pod linią miotły, takim, który znajduje się pomiędzy tymi samymi krawędziami wielokąta. Jest to pierwszy wierzchołek zastępujący ten wierzchołek w zmiennej pomocniczej dowolnej krawędzi. Za każdym razem, gdy zastępujemy wierzchołek w zmiennej pomocniczej, sprawdzamy, czy nie był on wierzchołkiem scalającym, a jeżeli był, dodajemy odcinek łączący obecnie rozpatrywany wierzchołek z zapisanym wierzchołkiem scalającym. Dzieje się tak również w przypadku, gdy obecnie rozpatrywanym wierzchołkiem jest wierzchołek dzielący. W takiej sytuacji pozbywamy się równocześnie wierzchołka dzielącego, jak i scalającego za pomocą jednej odcinka. Przykład dodawania odcinka wychodzącego z wierzchołka scalającego można zobaczyć na rysunku 4.6.

Sam algorytm przebiega w następujący sposób: na samym starcie algorytm tworzy mapę planarną (obiektu klasy PlanarMap) odpowiadają wielokątowi, którego poddział monotoniczny będzie wyznaczany, a także inne pomocnicze struktury potrzebne w trakcie głównej części algorytmu, czyli listę zdarzeń, słownik pomocniczy do przechowywania informacji o obecnie najniżej położonym wierzchołku nad linią miotły, dla każdej krawędzi wielokąta, słownik przechowujący informację o typach wierzchołków oraz obiekt klasy SlowTreeX, czyli drzewo przechowujące informację o krawędziach. Na począt-



Rysunek 4.6. Przykład dodawania odcinka wychodzącego z wierzchołka scalającego. Odcinek ten zostanie dodany w momencie rozpatrywania wierzchołka v_{m} [13].

ku zasadniczej części algorytmu przygotowujemy listę wydarzeń, czyli listę wierzchołków z przypisanymi typami. Typy wierzchołków przechowywane są w osobnym słowniku, gdzie kluczem jest wierzchołek, a wartością jego typ.

Następnie przechodzimy po wszystkich wierzchołkach wielokąta i dla każdego ustalamy jego typ, na podstawie sąsiadujących wierzchołków, po czym dodajemy go do listy wydarzeń informacje o wierzchołku, typie wydarzenia oraz krawędziach prowadzących od wierzchołka do sąsiadujących z nim wierzchołków. Na koniec sortujemy listę wydarzeń, ponieważ nie zmienia się ona w trakcie trwania algorytmu. Wydarzenia sortujemy względem współrzędnych y wierzchołków, a w przypadku takiej samej wartości sortujemy dodatkowo względem współrzędnych x. Dalsza część algorytmu polega na rozpatrywaniu po kolej wydarzeń z listy, wykonując operacje w zależności od typu wydarzenia. Po przejściu przez całą listę wydarzeń algorytm kończy się, a na wyjściu otrzymujemy mapę planarną przechowującą podział monotoniczny wejściowego wielokąta.

Rysunek 4.7 przedstawia wynikowy podział monotoniczny dla przykładowego wielokąta prostego.

Złożoność: Teoretyczna złożoność czasowa algorytmu wynosi $O(n \log n)$. W praktyce, tak samo jak w przypadku algorytmu triangulacji wielokąta y-monotonicznego, następuje pogorszenie złożoności spowodowane metodą planar_map.add_chord(segment).

Drugim elementem pogarszającym złożoność jest implementacja drzewa przechowującego odcinki. Użyta implementacja symuluje zachowanie optymalnego rozwiązania, jednak z gorszą złożonością czasową. Strukturą pozwalającą osiągnąć optymalną złożoność czasową algorytmu jest zmodyfikowane drzewo AVL.

Listing 4.5. Moduł partition1.

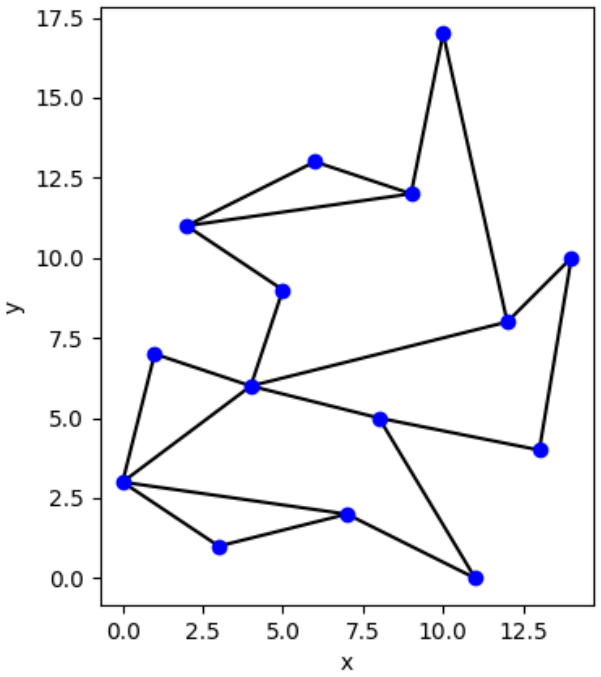
```
#!/usr/bin/env python3

try:
    range = xrange
except NameError:    # Python 3
    pass

#from planegeometry.structures.points import Point
from planegeometry.structures.segments import Segment
from planegeometry.structures.polygons import Polygon
from planegeometry.structures.planarmaps import PlanarMap
from planegeometry.algorithms.geomtools import orientation
from planegeometry.structures.events import Event
from planegeometry.structures.slowtrees import SlowTreeX

class YMonotonePartitionPM:

    def __init__(self, polygon):
        if polygon.orientation(test_is_simple=False) < 0:
            raise ValueError("clockwise orientation detected")
        self.point_list = polygon.point_list
        self.planar_map = PlanarMap(polygon)
        self.event_queue = []
        self.helper = dict()
        self.vertex_type = dict()
        self.sweep_line = SlowTreeX()

    def run(self):
        self._prepare_event_queue()
        for event in self.event_queue:
            if event.type == Event.START_VERTEX:
                self._handle_start_vertex(event)
            elif event.type == Event.END_VERTEX:
                self._handle_end_vertex(event)
            elif event.type == Event.SPLIT_VERTEX:
                self._handle_split_vertex(event)
            elif event.type == Event.MERGE_VERTEX:
                self._handle_merge_vertex(event)
            elif event.type == Event.REGULAR_VERTEX:
                self._handle_regular_vertex(event)
            else:
                raise ValueError("unknown event")
        del self.event_queue
        del self.sweep_line

    def find_outer_face(self):
        pt1 = self.point_list[0]
        pt2 = self.point_list[1]
        outer_face = self.planar_map.edge2face[Segment(pt1, pt2)]
        return outer_face

    def _prepare_event_queue(self):
        n = len(self.point_list)
        for i in range(n):
            pt1 = self.point_list[i]
            pt2 = self.point_list[(i + 1) % n]
```

```
            pt3 = self.point_list[(i + 2) % n]
            segment1, segment3 = self.planar_map.iteroutedges(pt2)
            if segment1.pt2 == pt3:
                segment1, segment3 = segment3, segment1
            if segment1.pt1 > segment1.pt2:
                segment1 = ~segment1
            if segment3.pt1 > segment3.pt2:
                segment3 = ~segment3

            if (pt2.y > pt1.y and pt2.y > pt3.y and
                orientation(pt1, pt2, pt3) > 0):    # /|  start vertex
                self.vertex_type[pt2] = Event.START_VERTEX
                self.event_queue.append(
                    Event(pt2, Event.START_VERTEX, segment1, segment3))
            elif (pt2.y < pt1.y and pt2.y < pt3.y and
                orientation(pt1, pt2, pt3) > 0):    # |/  end vertex
                self.vertex_type[pt2] = Event.END_VERTEX
                self.event_queue.append(
                    Event(pt2, Event.END_VERTEX, segment1, segment3))
            elif (pt2.y > pt1.y and pt2.y > pt3.y and
                orientation(pt1, pt2, pt3) < 0):    # /|  split vertex
                self.vertex_type[pt2] = Event.SPLIT_VERTEX
                self.event_queue.append(
                    Event(pt2, Event.SPLIT_VERTEX, segment1, segment3))
            elif (pt2.y < pt1.y and pt2.y < pt3.y and
                orientation(pt1, pt2, pt3) < 0):    # |/  merge vertex
                self.vertex_type[pt2] = Event.MERGE_VERTEX
                self.event_queue.append(
                    Event(pt2, Event.MERGE_VERTEX, segment1, segment3))
            else:    # regular vertex
                self.vertex_type[pt2] = Event.REGULAR_VERTEX
                self.event_queue.append(
                    Event(pt2, Event.REGULAR_VERTEX, segment1, segment3))
        self.event_queue.sort(key=lambda event: (-event.point.y, event.point.x))

    def _handle_start_vertex(self, event):
        segment1 = event.segment_above
        segment2 = event.segment_below
        self.sweep_line.insert(segment2)
        self.helper[segment2] = event.point

    def _handle_end_vertex(self, event):
        segment1 = event.segment_above
        segment2 = event.segment_below
        if self.vertex_type[self.helper[segment1]] == Event.MERGE_VERTEX:
            segment3 = Segment(self.helper[segment1], event.point)
            self.planar_map.add_chord(segment3)
        self.sweep_line.remove(segment1)

    def _handle_split_vertex(self, event):
        segment1 = event.segment_above
        segment2 = event.segment_below

        self.sweep_line.insert(segment1)
        node = self.sweep_line.predecessor(segment1)
        self.sweep_line.remove(segment1)
```

```
            segment3 = Segment(self.helper[node.value], event.point)
            self.planar_map.add_chord(segment3)

        self.sweep_line.insert(segment2)
        self.helper[segment2] = event.point

    def _handle_merge_vertex(self, event):
        segment1 = event.segment_above
        segment2 = event.segment_below
        if self.vertex_type[self.helper[segment1]] == Event.MERGE_VERTEX:
            segment3 = Segment(self.helper[segment1], event.point)
            self.planar_map.add_chord(segment3)
        node = self.sweep_line.predecessor(segment1)
        self.sweep_line.remove(segment1)
        if self.vertex_type[self.helper[node.value]] == Event.MERGE_VERTEX:
            segment3 = Segment(self.helper[node.value], event.point)
            self.planar_map.add_chord(segment3)
        self.helper[node.value] = event.point

    def _handle_regular_vertex(self, event):
        segment1 = event.segment_above
        segment2 = event.segment_below
        if max(segment1.pt1.y, segment1.pt2.y) > max(segment2.pt1.y,
                                                     segment2.pt2.y):
            # Wnetrze wielokata na prawo od event.point.
            if self.vertex_type[self.helper[segment1]] == Event.MERGE_VERTEX:
                segment3 = Segment(self.helper[segment1], event.point)
                self.planar_map.add_chord(segment3)
            self.sweep_line.remove(segment1)
            self.sweep_line.insert(segment2)
            self.helper[segment2] = event.point
        else:   # Wnetrze wielokata na lewo od event.point.
            self.sweep_line.insert(segment1)   # na chwile
            node = self.sweep_line.predecessor(segment1)
            self.sweep_line.remove(segment1)
            if self.vertex_type[self.helper[node.value]] == Event.MERGE_VERTEX:
                segment3 = Segment(self.helper[node.value], event.point)
                self.planar_map.add_chord(segment3)
            self.helper[node.value] = event.point
```

Listing 4.6. Przykład użycia algorytmu podziału monotonicznego wielokąta prostego

```
>>> from points import Point
>>> from polygons import Polygon
>>> from partition1 import YMonotonePartitionPM as Partition
>>> point_list = [Point(14, 10), Point(12, 8), Point(10, 17),
    Point(9, 12), Point(6, 13), Point(2, 11), Point(5, 9),
    Point(4, 6), Point(1, 7), Point(0, 3), Point(3, 1),
    Point(7, 2), Point(11, 0), Point(8, 5), Point(13, 4)]
>>> polygon = Polygon(*point_list)
>>> algorithm = Partition(polygon)
>>> algorithm.run()
>>> algorithm.planar_map.show()
```



Rysunek 4.7. Przykładowy podział monotoniczny wielokąta prostego.

4.4. Wyznaczanie kierunków monotoniczności wielokąta

Dla dowolnego wielokąta prostego przedstawimy algorytm wyznaczający wszystkie kierunki, dla których jest on monotoniczny. Algorytm przechodzi przez wszystkie wierzchołki wielokąta, sprawdzając, które z nich mogą mieć wpływ na kierunki monotoniczności, a następnie w odpowiedni sposób je rozpatrując. Alternatywny algorytm opisany został w pracy Preparata i Supowita [14].

Dane wejściowe: Obiekt klasy Polygon opisujący wielokąt prosty, którego kierunki monotoniczności chcemy wyznaczyć.

Dane wyjściowe: Lista przedziałów zabronionych kątów, lista przedziałów dozwolonych kątów oraz lista przedziałów wartości współczynnika nachylenia, definiująca możliwe nachylenia prostych, dla których wielokąt jest monotoniczny. Aby algorytm zwrócił listę przedziałów dozwolonych kątów przy wywołaniu metody run rozpoczynającej działanie algorytmu należy podać dodatkowy argument calculate_possible_angles z wartością True, a żeby algorytm zwrócił także listę przedziałów współczynników nachylenia, należy również podać argument calculate_slopes z wartością True.

Problem: Wyznaczanie kierunków monotoniczności wielokąta prostego.

Opis algorytmu: Algorytm rozpoczyna od sprawdzenia, czy podany na wejściu wielokąt jest wielokątem prostym, w przeciwnym wypadku rzucany jest wyjątek ValueError. W kolejnym kroku inicjalizujemy kolejno: pustą tablicę kątów globalnych wierzchołków wielokąta, pustą tablicę indeksów wierzchołków wklęsłych wielokąta, oraz puste tablice niedozwolonych par kątów, dozwolonych par kątów i par współczynników nachylenia, przechowujące dane wyjściowe. Następnie rozpoczyna się główna część algorytmu.

Na początku algorytm przechodzi przez wszystkie wierzchołki wielokąta i dla każdego wierzchołka oblicza jego kąt globalny. Aby to zrobić, odejmujemy od wartości y następnego wierzchołka występującego po aktualnie rozważanym wierzchołku wartość y obecnego wierzchołka i zapisujemy w zmiennej pomocniczej. To samo robimy dla wartości x obu wierzchołków. Posiadając obie wartości zapisane w zmiennych pomocniczych możemy wyznaczyć kąt globalny aktualnie rozważanego wierzchołka za pomocą funkcji atan2(), będącej częścią modułu math [15] języka Python. Wyznaczony kąt dodajemy do tablicy kątów globalnych.

Później po raz kolejny przechodzimy pętlą przez wszystkie wierzchołki wielokąta, tym razem w celu odnalezienia wszystkich wierzchołków wklęsłych, ponieważ to one określają, dla jakich kierunków wielokąt nie będzie monotoniczny. By tego dokonać, musimy wyznaczyć kąty wewnętrzne wierzchołków. Postępujemy w sposób częściowo podobny jak przy wyznaczaniu kątów globalnych. Od wyznaczonej wartości kąta globalnego wierzchołka następnego po aktualnie rozważanym wierzchołku odejmujemy kąt globalny aktualnie rozważanego wierzchołka i zapisujemy w zmiennej pomocniczej. W przypadku, gdy wartość kąta globalnego następnego wierzchołka jest mniejsza od kąta globalnego aktualnego wierzchołka, do wyznaczonego kąta wewnętrznego dodajemy 2π. Na koniec sprawdzamy, czy obliczony kąt wewnętrzny jest większy od π, a jeżeli tak, mamy do czynienia z wierzchołkiem wklęsłym, więc dodajemy jego indeks w liście wierzchołków do listy indeksów wierzchołków wklęsłych.

Następnym krokiem jest wyznaczenie zakresów niedozwolonych kątów. Kąty te określają jakie proste przecinają wielokąt w więcej niż dwóch miejscach, czyli proste prostopadłe do kierunków, które na pewno nie są monotoniczne. Nie musimy sprawdzać całego zakresu $[0, 2\pi]$, ponieważ dla tego zastosowania kąt α jest tożsamy z kątem $\alpha + \pi$, wystarczy więc wyznaczyć zakresy z przedziału $[0, \pi]$. Przechodzimy pętlą po tablicy indeksów wierzchołków wklęsłych. Dla każdego wierzchołka wklęsłego zapisujemy w zmiennej pomocniczej $a1$ resztę z dzielenia jego kąta globalnego przez π. To samo robimy dla następnego wierzchołka wielokąta występującego po aktualnie rozważanym wierzchołku i zapisujemy w zmiennej $a2$. Te dwie wyznaczone wartości definiują przedział niedozwolonych kątów. W przypadku, gdy wartość wyznaczona dla następnego wierzchołka jest większa od wartości wyznaczonej dla obecnego wierzchołka, dodajemy do listy niedozwolonych kątów przedział $[a2, a1]$. W przeciwnym wypadku dodajemy do listy niedozwolonych kątów dwa przedziały: $[0, a1]$ i $[a2, \pi]$. Po przejściu przez całą listę wierzchołków wklęsłych otrzymujemy listę niedozwolonych kątów, która pozwala na wyznaczenie wszystkich kierunków monotoniczności, dlatego

algorytm kończy się w tym miejscu, jeśli metoda run została wywołana bez dodatkowych parametrów.

W sytuacji, gdy algorytm uruchomiony został z dodatkowym parametrem calculate_possible_angles ustawionym na True, na wyjściu otrzymamy również listę dozwolonych kątów, która w przeciwieństwie do listy niedozwolonych kątów nie przechowuje zduplikowanych kątów. W przypadku listy niedozwolonych kątów każdy wierzchołek wklęsły dodaje do listy swój przedział lub przedziały, więc jeżeli wiele wierzchołków wyklucza te same kąty, informacja ta będzie zduplikowana w liście. Aby wyznaczyć listę dozwolonych kątów po głównej części algorytmu wykonujemy jeszcze jeden krok. Na początku dodajemy do pustej listy dozwolonych kątów przedział $[0, \pi]$. Następnie przechodzimy przez listę niedozwolonych kątów i dla każdego elementu postępujemy w następujący sposób. Tworzymy pustą pomocniczą listę dozwolonych kątów. Później przechodzimy pętlą po wszystkich elementach aktualnej listy dozwolonych kątów i dla każdego zapisanego w niej zakresu: w przypadku, gdy aktualnie rozważany zakres niedozwolonych kątów nie ma części wspólnej z aktualnie rozważanym zakresem dozwolonych kątów, dodajemy do nowej listy dozwolonych kątów aktualnie rozważany zakres dozwolonych kątów. W innym wypadku, jeżeli kąt początkowy aktualnie rozważanego zakresu dozwolonych kątów jest mniejszy niż kąt początkowy aktualnie rozważanego zakresu kątów niedozwolonych, dodajemy do nowej listy dozwolonych kątów zakres od kąta początkowego aktualnie rozważanego zakresu dozwolonych kątów do kąta początkowego aktualnie rozważanego zakresu niedozwolonych kątów. W analogiczny sposób sprawdzamy, czy kąt końcowy aktualnie rozważanego zakresu dozwolonych kątów jest większy niż kąt końcowy aktualnie rozważanego zakresu kątów niedozwolonych, a jeśli tak, dodajemy do nowej listy dozwolonych kątów zakres od kąta końcowego aktualnie rozważanego zakresu niedozwolonych kątów do kąta końcowego aktualnie rozważanego zakresu dozwolonych kątów. Po przejściu przez całą listę dozwolonych kątów podmieniamy ją na listę pomocniczą, do której dodawaliśmy zakresy dozwolonych kątów i przechodzimy kolejnego do zakresu niedozwolonych kątów. Po rozpatrzeniu całej listy niedozwolonych kątów otrzymujemy na wyjściu listę dozwolonych kątów, która nie zawiera zduplikowanych informacji.

Program pozwala także na uruchomienie z parametrem calculate_slopes. W przypadku wartości True, po wyznaczeniu listy dozwolonych kątów algorytm wyznacza również listę przedziałów wartości współczynnika nachylenia prostych, dla których wielokąt jest monotoniczny. Aby tego dokonać, przechodzimy przez listę dozwolonych kątów i dla każdego zakresu wyznaczamy zakres współczynników nachylenia za pomocą funkcji przyjmującej jako argument kąt, a zwracającej wartość nachylenia, po czym dodajemy ten zakres do listy współczynników nachylenia.

Rysunek 4.8 przedstawia wyznaczone kierunki monotoniczności przykładowego wielokąta.

Złożoność: Złożoność czasowa głównej części algorytmu wynosi $O(n)$ i zgadza się z teoretyczną.

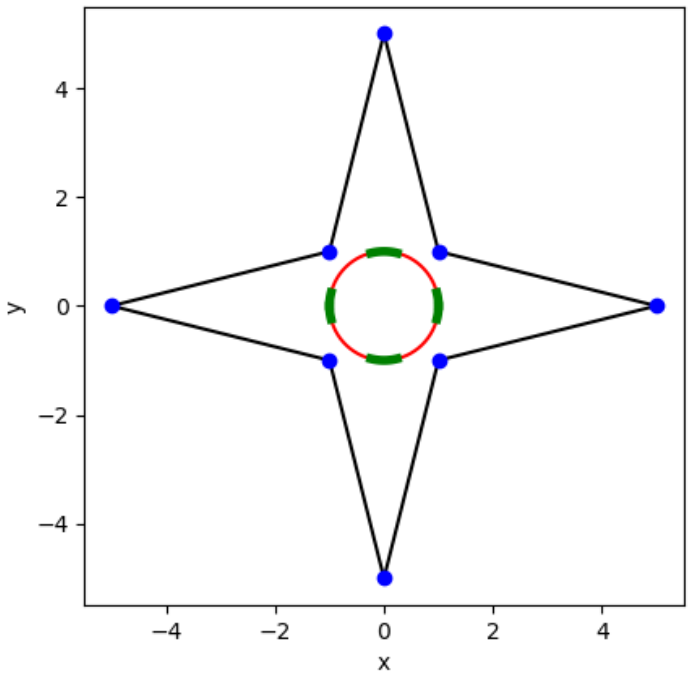
Listing 4.7. Moduł find_monotone_directions.

```
#!/usr/bin/env python3

try:
    range = xrange
except NameError:  # Python 3
    pass

import math

class FindMonotoneDirections:

    def __init__(self, polygon):
        if polygon.orientation(test_is_simple=False) < 0:
            raise ValueError("clockwise orientation detected")

        self.point_list = polygon.point_list
        self.n = len(self.point_list)

        self.absolute_angles = []
        self.concave_vertices_indices = []
        self.forbidden_angles_list = []  # for pairs of forbidden angle ranges
        self.possible_angles_range = []  # for pairs of possible angle boundaries
        self.result_monotone_range = []  # for pairs of slope boundaries

    def run(self, calculate_possible_angles=False, calculate_slopes=False):
        self.calculate_absolute_angles()
        self.find_concave_vertices()
        self.find_forbidden_angles()

        if calculate_possible_angles:
            self.find_possible_angles()
            if calculate_slopes:
                self.find_slope_ranges()

    def calculate_absolute_angles(self):
        for i in range(self.n):
            p1 = self.point_list[i]
            p2 = self.point_list[(i + 1) % self.n]
            p = p2 - p1
            self.absolute_angles.append(math.atan2(p.y, p.x))

    def find_concave_vertices(self):
        for i in range(self.n):
            a1 = self.absolute_angles[i]
            a2 = self.absolute_angles[(i + 1) % self.n]
            b = a2 - a1
            if a2 < a1:
                b += math.pi * 2
            if b > math.pi:
                self.concave_vertices_indices.append(i)

    def find_forbidden_angles(self):
        if self.concave_vertices_indices:
            for index in self.concave_vertices_indices:
                a1 = self.absolute_angles[index] % math.pi
                a2 = self.absolute_angles[(index + 1) % self.n] % math.pi
```

```
                if a2 < a1:
                    self.forbidden_angles_list.append((a2, a1))
                elif a1 < a2:
                    self.forbidden_angles_list.append((0, a1))
                    self.forbidden_angles_list.append((a2, math.pi))

    def find_possible_angles(self):
        self.possible_angles_range.append((0, math.pi))
        for start, end in self.forbidden_angles_list:
            self.exclude_range(start, end)

    def find_slope_ranges(self):
        if self.possible_angles_range:
            for start, end in self.possible_angles_range:
                slope_start = self.get_correct_slope_from_angle(start)
                slope_end = self.get_correct_slope_from_angle(end)
                self.result_monotone_range.append((slope_start, slope_end))

    def exclude_range(self, exclusion_start, exclusion_end):
        new_allowed_ranges = []
        for start, end in self.possible_angles_range:
            if exclusion_end < start or exclusion_start > end:
                new_allowed_ranges.append((start, end))
            else:
                if start < exclusion_start:
                    new_allowed_ranges.append((start, exclusion_start))
                if end > exclusion_end:
                    new_allowed_ranges.append((exclusion_end, end))

        self.possible_angles_range = new_allowed_ranges

    def get_correct_slope_from_angle(self, angle):
        if angle == 0:
            return math.inf
        elif angle == math.inf:
            return 0
        else:
            return -1 / math.tan(angle)
```

Listing 4.8. Przykład użycia algorytmu wyznaczającego kierunki monotoniczności wielokąta

```
>>> from points import Point
>>> from polygons import Polygon
>>> from find_monotone_directions import FindMonotoneDirections as Directions
>>> point_list = [Point(0, 5), Point(-1, 1), Point(-5, 0), Point(-1, -1),
    Point(0, -5), Point(1, -1), Point(5, 0), Point(1, 1)]
>>> polygon = Polygon(*point_list)
>>> algorithm = Directions(polygon)
>>> algorithm.run(True, True)
```



Rysunek 4.8. Wyznaczenie kierunków monotoniczności przykładowego wielokąta. Na zielono zaznaczone kierunki, dla których wielokąt jest monotoniczny, a na czerwono kierunki, dla których wielokąt nie jest monotoniczny.

4.5. Triangulacja wielokąta wypukłego z minimalizacją długości cięciw

Dla dowolnego wielokąta wypukłego przedstawimy jego triangulację z minimalizacją długości cięciw przy użyciu map planarnych. Algorytm nie dopuszcza występowania wierzchołków z kątem wewnętrznym 180°. Przedstawione zostaną dwie implementacje tego algorytmu, jedna wyznacza triangulację w sposób rekurencyjny, a druga w sposób iteracyjny. W przypadku obu wersji stosuje się technikę programowania dynamicznego. Algorytm powstał w oparciu o źródło [16].

Dane wejściowe: Obiekt klasy Polygon opisujący wielokąt wypukły, którego triangulację chcemy wyznaczyć.

Dane wyjściowe: Obiekt klasy PlanarMap przechowujący wyznaczoną triangulację wielokąta.

Problem: Triangulacja wielokąta wypukłego z minimalizacją długości cięciw.

Opis algorytmu: Oba algorytmy rozpoczynają się od utworzenia mapy planarnej (obiektu klasy PlanarMap) odpowiadającej wejściowemu wielokątowi, a także od inicjalizacji dwuwymiarowej tablicy, której każdy element przechowuje informację o minimalnym koszcie triangulacji wielokąta tworzonego przez wierzchołki o indeksach odpowiadających danej komórce tablicy. Przykładowo, element tablicy o indeksach [2][5] przechowuje minimalny koszt triangulacji wielokąta tworzonego przez wierzchołki 2, 3, 4 i 5. Każdy element przechowuje dodatkowo informację o indeksie wierzchołka, dla którego odnaleziony został minimalny koszt triangulacji danego wielokąta, co jest potrzebne do końcowego odtworzenia triangulacji. Funkcją kosztu przyjętą w celu minimalizacji długości cięciw jest odległość euklidesowa, natomiast główna część algorytmu zadziała dla dowolnej funkcji kosztu.

Po stworzeniu mapy planarnej i zainicjalizowaniu dwuwymiarowej tablicy kosztów rozpoczyna się zasadnicza część algorytmu. W przypadku algorytmu rekurencyjnego wywołujemy metodę obliczającą koszt triangulacji, jako argumenty podając pierwszy i ostatni wierzchołek wielokąta (chcemy obliczyć minimalny koszt triangulacji dla całego wielokąta). Metoda ta działa następująco: na początku sprawdzamy, czy podane wierzchołki nie są wierzchołkami sąsiednimi, a jeśli są, to mamy do czynienia z odcinkiem, a nie wielokątem, więc w takim przypadku koszt triangulacji wynosi 0. W przeciwnym wypadku sprawdzamy, czy w tablicy kosztów nie został już wyznaczony koszt dla podanego wielokąta, a jeżeli został, nie musimy go wyznaczać na nowo. Jeśli koszt dla aktualnie rozważanego wielokąta nie został jeszcze zapisany w tablicy kosztów, musimy go obliczyć.

Aby tego dokonać, na początku tworzymy dwie zmienne pomocnicze, z których jedna przechowywać będzie najmniejszy znaleziony koszt triangulacji tego wielokąta, a druga indeks wierzchołka, dla którego minimalny koszt został wyznaczony. Następnie, dla każdego wierzchołka pomiędzy wierzchołkiem startowym a końcowym sprawdzanego wielokąta obliczamy koszt triangulacji. Robimy to w sposób następujący: obliczamy funkcję kosztu dla trójkąta utworzonego przez wierzchołek startowy, wierzchołek końcowy i aktualnie rozważany wierzchołek między nimi obecnie rozpatrywanego wielokąta, dodając do tego minimalny koszt triangulacji wielokąta, dla którego wierzchołkiem startowym jest wierzchołek startowy obecnie rozpatrywanego wielokąta, a końcowym aktualnie rozważany wierzchołek oraz minimalny koszt triangulacji wielokąta, dla którego wierzchołkiem startowym jest aktualnie rozważany wierzchołek, a końcowym wierzchołek końcowy obecnie rozpatrywanego wielokąta (w tym miejscu następuje rekurencyjne wywołanie opisywanej metody). Jeżeli obliczony w ten sposób koszt jest niższy od kosztu zapisanego w zmiennej pomocniczej przechowującej minimalny koszt, przypisujemy go do zmiennej, oraz zapisujemy do zmiennej pomocniczej przechowującej indeks optymalnego wierzchołka indeks aktualnie rozważanego wierzchołka. Po sprawdzeniu wszystkich wierzchołków pomiędzy wierzchołkiem startowym a końcowym wielokąta zapisujemy w tablicy kosztów ostateczną wartość zmiennej przechowującej minimalny koszt triangulacji rozważanego wielokąta wraz z indeksem optymalnego wierzchołka.

Wywołanie tej metody dla pierwszego i ostatniego wierzchołka wielokąta oblicza triangulację całego wielokąta, uzupełniając tablicę kosztów infor-

macjami o koszcie, a także formie optymalnego rozwiązania. Jednak aby dodać wymagane krawędzie do mapy planarnej przechowującej ostateczne rozwiązanie, musimy w odpowiedni sposób wyciągnąć je z wyznaczonej tablicy kosztów. Tutaj również stosujemy podejście rekurencyjne. Do wybrania odpowiednich trójkątów stosujemy rekurencyjną metodę, przyjmującą jako argumenty: wierzchołek startowy i wierzchołek końcowy wielokąta, którego triangulacje chcemy otrzymać, oraz zwracającą tablicę trójek zawierających indeksy punktów tworzących trójkąt. Na początku tworzymy pustą tablicę do przechowywania trójek. Następnie, podobnie jak w przypadku metody wyznaczającej koszt triangulacji sprawdzamy, czy podane wierzchołki nie są wierzchołkami sąsiednimi, a jeśli są, zwracamy pustą tablicę. To samo robimy w przypadku, gdy w tablicy kosztów nie posiadamy informacji o optymalnym wierzchołku dla podanego wielokąta. W innym wypadku dodajmy do tablicy trójkę indeksów składającą się z indeksu wierzchołka startowego aktualnie rozważanego wielokąta, indeksu wierzchołka zapisanego jak optymalny w tablicy kosztów, oraz indeksu wierzchołka końcowego aktualnie rozważanego wielokąta. Wywołujemy rekurencyjnie opisywaną metodę, podając jako argumenty: wierzchołek startowy rozpatrywanego wielokąta i wierzchołek optymalny z tablicy kosztów oraz drugi raz podając jako argumenty: wierzchołek optymalny i wierzchołek końcowy rozpatrywanego wielokąta. Rezultaty rekurencyjnych wywołań również dodajemy do tablicy trójek, którą następnie zwracamy.

Podobnie jak przy obliczaniu kosztu, wywołujemy metodę zwracającą triangulację dla pierwszego i ostatniego wierzchołka wielokąta. Otrzymujemy tablicę trójek, zawierającą informację o wszystkich trójkątach triangulacji. Na koniec przechodzimy przez wszystkie elementy tablicy i dla każdego trójkąta dodajemy do mapy planarnej każdą jego krawędź, której nie ma jeszcze w mapie. W ten sposób na wyjściu otrzymujemy mapę planarną przechowującą obliczoną triangulację z minimalną długości cięciw wejściowego wielokąta.

Wersja iteracyjna algorytmu różni się w dwóch miejscach. Metoda wyznaczająca tablicę kosztów zamiast rekurencji w odpowiedni sposób przechodzi po elementach tablicy, aby wyliczyć kolejne koszty. W tym celu stosujemy potrójną pętlę. Analogicznie jak w przypadku wersji rekurencyjnej, dla każdego elementu tablicy kosztów, obliczamy oraz porównujemy koszty możliwych podziałów i zapisujemy w tablicy kosztów najmniejszy wraz z indeksem optymalnego wierzchołka. Drugą różnicą jest sposób wyciągania z tablicy kosztów informacji o ostatecznej formie triangulacji. Zamiast rekurencji stosujemy w tym przypadku stos przechowujący pary indeksów wyznaczające elementy tablicy kosztów, których potrzebujemy do utworzenia optymalnej triangulacji. Na starcie tworzymy również pustą tablicę trójek indeksów punktów tworzących trójkąt oraz dodajemy na stos pierwszą parę indeksów, czyli indeksy pierwszego i ostatniego wierzchołka wielokąta. Później postępujemy następująco: dopóki stos nie jest pusty ściągamy z niego parę indeksów i jeśli nie są to indeksy sąsiadujących wierzchołków oraz posiadamy w tablicy kosztów zapisany indeks optymalnego wierzchołka, dodajemy na stos dwie nowe pary indeksów, składającą się odpowiednio z pierwszego indeksu aktualnej pary i indeksu optymalnego wierzchołka oraz parę składającą się z indeksu

optymalnego wierzchołka i drugiego indeksu aktualnej pary. Dodajemy również do tablicy trójek trójkę złożoną z pierwszego indeksu aktualnej pary, indeksu optymalnego wierzchołka oraz z drugiego indeksu aktualnej pary. W momencie przetworzenia całego stosu otrzymujemy tablicę trójek zawierającą informację o wszystkich trójkątach triangulacji. W ten sam sposób co w wersji rekurencyjnej wykorzystujemy ją, aby dodać do mapy planarnej brakujące krawędzie triangulacji.

Rysunek 4.9 przedstawia wynikową triangulację z minimalną długością cięciw dla przykładowego wielokąta wypukłego.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n^3)$, zarówno w przypadku algorytmu w wersji rekurencyjnej, jak i wersji iteracyjnej. W praktyce algorytm w wersji iteracyjnej działa trochę szybciej, ale jego złożoność czasowa nadal wynosi $O(n^3)$.

Uwagi: Metodę programowania dynamicznego można zastosować do triangulacji ogólnego wielokąta prostego z minimalizacją długości cięciw [17]. Dodatkowa trudność wynika z faktu, że nie każdy odcinek łączący wierzchołki wielokąta prostego będzie zawarty w całości wewnątrz wielokąta. Łatwiej jest uogólnić triangulację na wielokąty wypukłe z dozwolonymi kątami wewnętrznymi równymi 180 stopni. Wystarczy rozpoznać zdegenerowane trójkąty (trzy wierzchołki współliniowe) i przyporządkować im nieskończoną wartość funkcji kosztu, przez co nie będą brane pod uwagę.

Listing 4.9. Moduł triangulation_min_chord_iterative.

```
#!/usr/bin/env python3

try:
    range = xrange
except NameError:  # Python 3
    pass

from planegeometry.structures.segments import Segment
from planegeometry.structures.planarmaps import PlanarMap


class MinimumChordTriangulation:

    def __init__(self, polygon):
        if polygon.orientation(test_is_simple=False) < 0:
            raise ValueError("clockwise orientation detected")

        self.point_list = polygon.point_list
        self.n = len(self.point_list)
        self.planar_map = PlanarMap(polygon)

        self.cost_matrix = [[[-1 for k in range(2)]
            for j in range(self.n)] for i in range(self.n)]
            # 2d array of pairs (cost, index)

    def run(self):
        self.calculate_cost()
        triangulation = self.get_triangulation()
```

```
        for (i,j,k) in triangulation:
            segment = Segment(self.point_list[i], self.point_list[j])
            if not self.planar_map.has_edge(segment):
                self.planar_map.add_chord(segment)

            segment = Segment(self.point_list[j], self.point_list[k])
            if not self.planar_map.has_edge(segment):
                self.planar_map.add_chord(segment)

    def calculate_cost(self):
        # Przypadek bazowy - pojedyncza krawedz.
        for i in range(0, self.n-1):
            self.cost_matrix[i][i+1][0] = 0
        # Dalsza propagacja obliczen.
        for skip in range(2, self.n):
            for i in range(self.n - skip):
                j = i + skip

                min_cost = float('inf')
                index = -1

                for k in range(i + 1, j):
                    cost = (self.cost_matrix[i][k][0]
                        + self.cost_matrix[k][j][0]
                        + self.cost_function(i, k, j))
                    if cost < min_cost:
                        min_cost = cost
                        index = k

                self.cost_matrix[i][j][0] = min_cost
                self.cost_matrix[i][j][1] = index

    def cost_function(self, i, j, k):
        a = self.point_list[i]
        b = self.point_list[j]
        c = self.point_list[k]
        return (a - b).length() + (b - c).length() + (a - c).length()

    def get_triangulation(self):
        stack = [(0, self.n - 1)]
        triangulation = []

        while stack:
            i, j = stack.pop()
            if j - i < 2:
                continue

            k = self.cost_matrix[i][j][1]
            if k != -1:
                stack.append((i, k))
                stack.append((k, j))
                triangulation.append((i, k, j))

        return triangulation
```

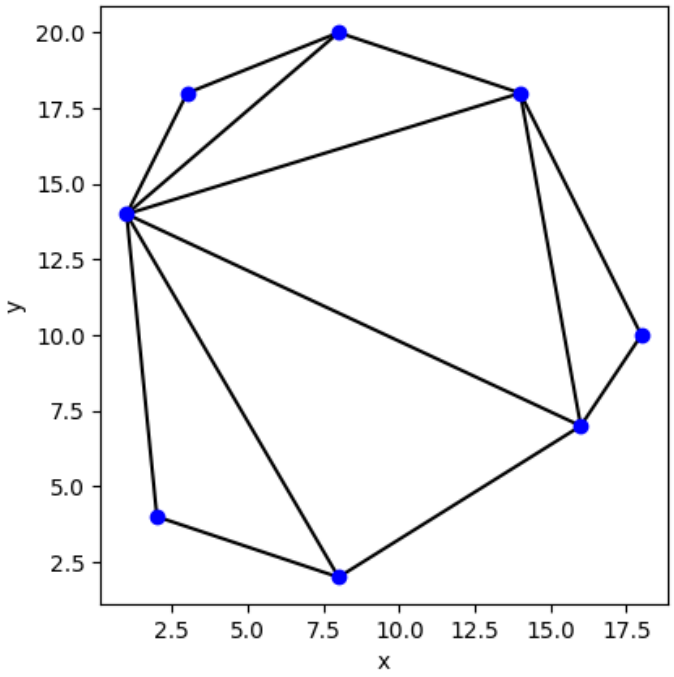
Listing 4.10. Moduł triangulation_min_chord_recursive.

```
#!/usr/bin/env python3

try:
    range = xrange
except NameError:  # Python 3
    pass

from planegeometry.structures.segments import Segment
from planegeometry.structures.planarmaps import PlanarMap


class MinimumChordTriangulation:

    def __init__(self, polygon):
        if polygon.orientation(test_is_simple=False) < 0:
            raise ValueError("clockwise orientation detected")

        self.point_list = polygon.point_list
        self.n = len(self.point_list)
        self.planar_map = PlanarMap(polygon)

        self.cost_matrix = [[[-1 for k in range(2)]
            for j in range(self.n)] for i in range(self.n)]
            # 2d array of pairs (cost, index)

    def run(self):
        self.calculate_cost(0, self.n - 1)
        triangulation = self.get_triangulation(0, self.n - 1)

        for (i,j,k) in triangulation:
            segment = Segment(self.point_list[i], self.point_list[j])
            if not self.planar_map.has_edge(segment):
                self.planar_map.add_chord(segment)

            segment = Segment(self.point_list[j], self.point_list[k])
            if not self.planar_map.has_edge(segment):
                self.planar_map.add_chord(segment)

    def calculate_cost(self, i, j):
        if j - i < 2:
            return 0

        if self.cost_matrix[i][j][0] != -1:
            return self.cost_matrix[i][j][0]

        min_cost = float('inf')
        index = -1
        for k in range(i + 1, j):
            cost = (self.calculate_cost(i, k)
                + self.calculate_cost(k, j)
                + self.cost_function(i, k, j))
            if cost < min_cost:
                min_cost = cost
                index = k

        self.cost_matrix[i][j][0] = min_cost
```

```
            self.cost_matrix[i][j][1] = index

            return min_cost

        def cost_function(self, i, j, k):
            a = self.point_list[i]
            b = self.point_list[j]
            c = self.point_list[k]
            return (a - b).length() + (b - c).length() + (a - c).length()

        def get_triangulation(self, i, j):
            triangulation = []

            if j - i < 2:
                return triangulation

            k = self.cost_matrix[i][j][1]
            if k == -1:
                return triangulation

            triangulation.append((i, k, j))
            return (triangulation + self.get_triangulation(i, k)
                    + self.get_triangulation(k, j))
```

Listing 4.11. Przykład użycia triangulacji z minimalizacją długości cięciw.

```
>>> from points import Point
>>> from polygons import Polygon
>>> from triangulation_min_chord_iterative1 import MinimumChordTriangulation
    as Triangulation
>>> point_list = [Point(8, 20), Point(3, 18), Point(1, 14), Point(2, 4),
    Point(8, 2), Point(16, 7), Point(18, 10), Point(14, 18)]
>>> polygon = Polygon(*point_list)
>>> algorithm = Triangulation(polygon)
>>> algorithm.run()
>>> algorithm.planar_map.show()
```



Rysunek 4.9. Przykładowa triangulacja wielokąta wypukłego z minimalną długością cięciw.

5. Podsumowanie

Praca zawiera analizę i implementację wybranych algorytmów dla wielokątów monotonicznych, a w szczególności dla wielokątów wypukłych. Algorytmy triangulacji wachlarzowej wielokąta wypukłego i triangulacji wielokąta y-monotonicznego zaimplementowano z wykorzystaniem map planarnych. Wcześniejsza implementacja przechowywała trójkąty w strukturze kolekcji, która nie obsługuje ogólniejszych podziałów wielokąta i trudniej z niej uzyskać informacje o topologii podziału. Testy wydajnościowe pokazały, że użycie map planarnych psuje złożoność $O(n)$ z powodu czasochłonnej operacji lokalizacji miejsca wstawienia nowej krawędzi pomiędzy stare krawędzie wychodzące z danego wierzchołka. Przygotowano dwie nowe klasy, które przechowują podział w strukturze grafu abstrakcyjnego. Otrzymano złożoność $O(n)$, ale informacja o topologii podziału wielokąta nie jest dostępna wprost.

W pracy analizowano algorytm wyznaczania kierunków monotoniczności danego wielokąta prostego. Zabronione kierunki są wyznaczone przez wierzchołki wklęsłe wielokąta. Każdy wierzchołek wklęsły wprowadza pewien przedział zabronionych kątów, które należy połączyć w celu otrzymania pełnego zestawu zabronionych kątów. W implementacji ujawniają się trudności charakterystyczne dla topologii okręgu: musimy gdzieś wykonać cięcie na okręgu, aby go sparametryzować.

Zaimplementowano algorytm podziału wielokąta prostego na części monotoniczne. Algorytm wykorzystuje technikę zamiatania płaszczyzny, a podział wielokąta przechowywany jest jako mapa planarna. Kod jest zawarty w klasie YMonotonePartitionPM, a wersja z grafem abstrakcyjnym w klasie YMonotonePartitionGraph. Następnym krokiem może być triangulacja wskazanej części monotonicznej lub wszystkich części monotonicznych, co prowadzi do pełnej triangulacji wielokąta prostego.

Rozważono problem triangulacji wielokąta wypukłego przy jednoczesnej minimalizacji sumy długości cięciw (funkcja kosztu). Rozwiązanie bazuje na metodzie programowania dynamicznego, ponieważ dany wielokąt można podzielić na trójkąt i dwa mniejsze wielokąty wypukłe. Bazowy przypadek to pojedyncza krawędź o koszcie zerowym. Koszt trójkąta to suma długości jego boków. Zaprezentowano dwa rodzaje implementacji: rekurencyjną i iteracyjną.

Wielokąty monotoniczne są jakby pośrednim ogniwem pomiędzy wielokątami wypukłymi, a ogólnymi wielokątami prostymi. Ich użycie może pomóc w rozwiązywaniu problemów z geometrii obliczeniowej. Mamy nadzieję, że przygotowane w ramach tej pracy narzędzia ułatwią pracę z wielokątami monotonicznymi i zainspirują do tworzenia nowych, wydajnych i praktycznych algorytmów.

A. Testy algorytmów

W tym dodatku opisane zostaną wyniki przeprowadzonych testów wydajnościowych mających na celu potwierdzenie teoretycznej złożoności obliczeniowej przedstawionych w pracy algorytmów. Do wyznaczenia pomiarów czasu wykonywania algorytmów użyty został moduł języka Python timeit [18]. Moduł ten pozwala na proste testowanie fragmentów kodu, dzięki czemu jest bardzo przydatny do sprawdzania wydajności.

Testy przeprowadzone zostały na komputerze z procesorem Intel Core i5 3.9GHz.

A.1. Testy triangulacji wachlarzowej wielokąta wypukłego

Test implementacji triangulacji wachlarzowej z wykorzystaniem grafu do przechowania informacji o triangulacji potwierdza, że jej złożoność obliczeniowa zgadza się z teoretyczną złożonością $O(n)$. Dowodem na to jest wykres A.1. W przypadku implementacji wykorzystującej mapę planarną, złożoność obliczeniowa algorytmu jest gorsza niż teoretyczna, co jest ceną za posiadanie informacji o strukturze topologicznej triangulacji. Obrazuje to wykres A.2. Testy wykonano dla wielokąta wypukłego przypominającego trapez z wieloma punktami na równoległych bokach.

A.2. Testy triangulacji wielokąta y-monotonicznego

Test implementacji triangulacji wielokąta y-monotonicznego z wykorzystaniem grafu do przechowania informacji o triangulacji potwierdza, że jej złożoność obliczeniowa zgadza się z teoretyczną złożonością $O(n)$. Dowodzi temu wykres A.3. W przypadku implementacji wykorzystującej mapę planarną, analogicznie jak w przypadku algorytmu triangulacji wachlarzowej, złożoność obliczeniowa algorytmu jest gorsza niż teoretyczna, co jest ceną za posiadanie informacji o strukturze topologicznej triangulacji. Można to zauważyć na wykresie A.4. Testy wykonano dla wielokąta wypukłego przypominającego trapez z wieloma punktami na równoległych bokach.

A.3. Testy podziału monotonicznego wielokąta prostego

Do testów przygotowano generator wielokątów przypominających pas z ząbkowanymi brzegami. Liczba wszystkich wierzchołków, krawędzi, wierzchołków rozdzielających i wierzchołków scalających rośnie liniowo wraz z pa-

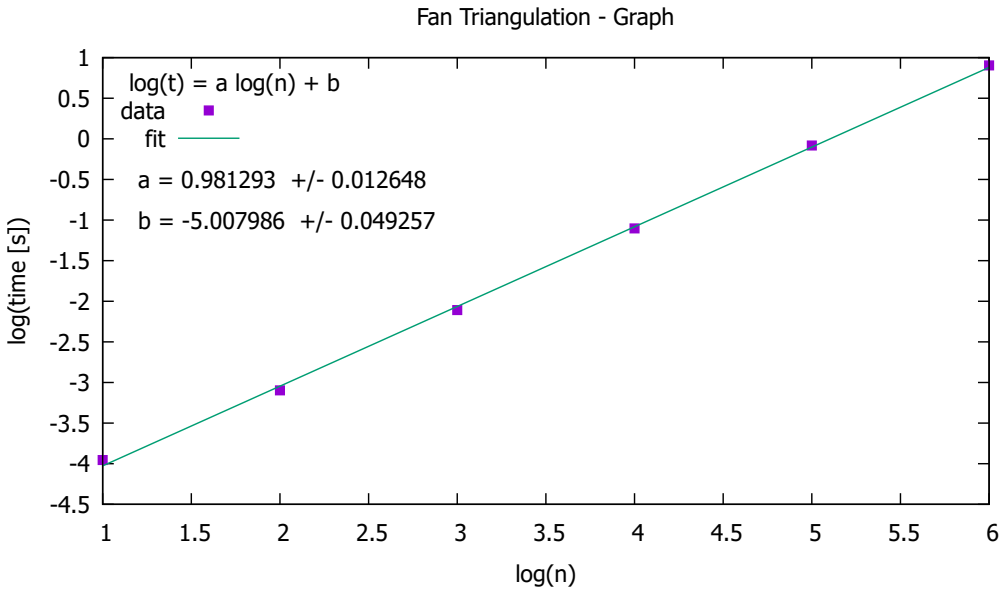
rametrem generatora. Test implementacji podziału monotonicznego wielokąta z wykorzystaniem grafu do przechowania informacji o triangulacji potwierdza, że jej złożoność obliczeniowa zgadza się z teoretyczną złożonością $O(n \log n)$. Potwierdza to wykres A.5. Tak samo, jak w przypadku algorytmów triangulacji wachlarzowej wielokąta wypukłego i triangulacji wielokąta y-monotonicznego, złożoność obliczeniowa algorytmu wykorzystującego mapę planarną do przechowywania podziału jest gorsza niż teoretyczna w zamian za informacje o strukturze topologicznej podziału. Jest to widoczne na wykresie A.6.

A.4. Testy triangulacji z minimalizacją długości cięciw

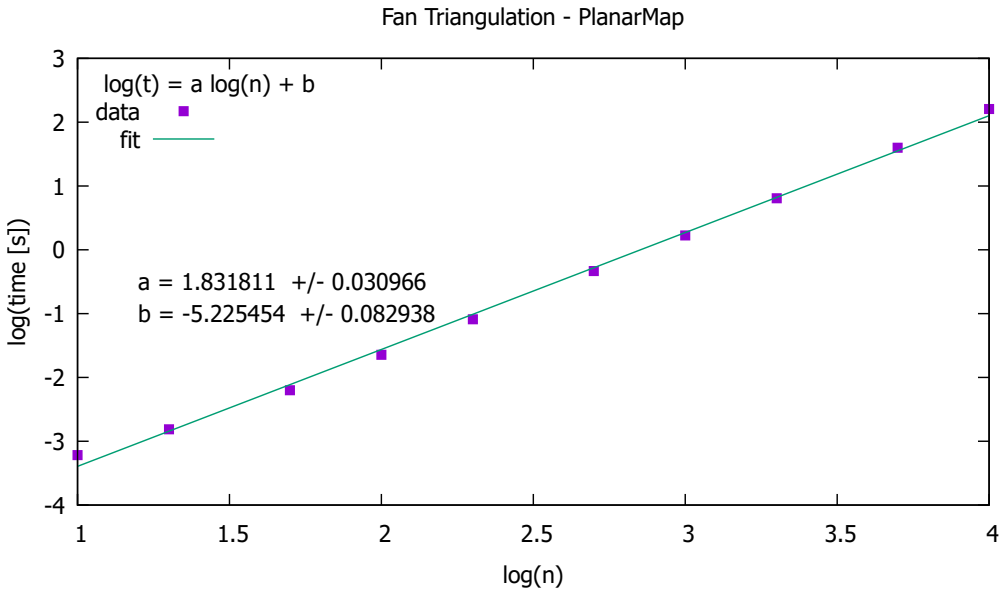
Testy implementacji triangulacji z minimalizacją długości cięciw potwierdzają, że zarówno w wersji iteracyjnej, jak i rekurencyjnej, złożoność obliczeniowa zgadza się z teoretyczną złożonością $O(n^3)$. Złożoność ta jest taka sama zarówno dla implementacji przechowujących wyjściową triangulację w grafie jak i w mapie planarnej, czego dowodzą wykresy A.7, A.8, A.9, A.10.

A.5. Test algorytmu wyznaczania kierunków monotoniczności

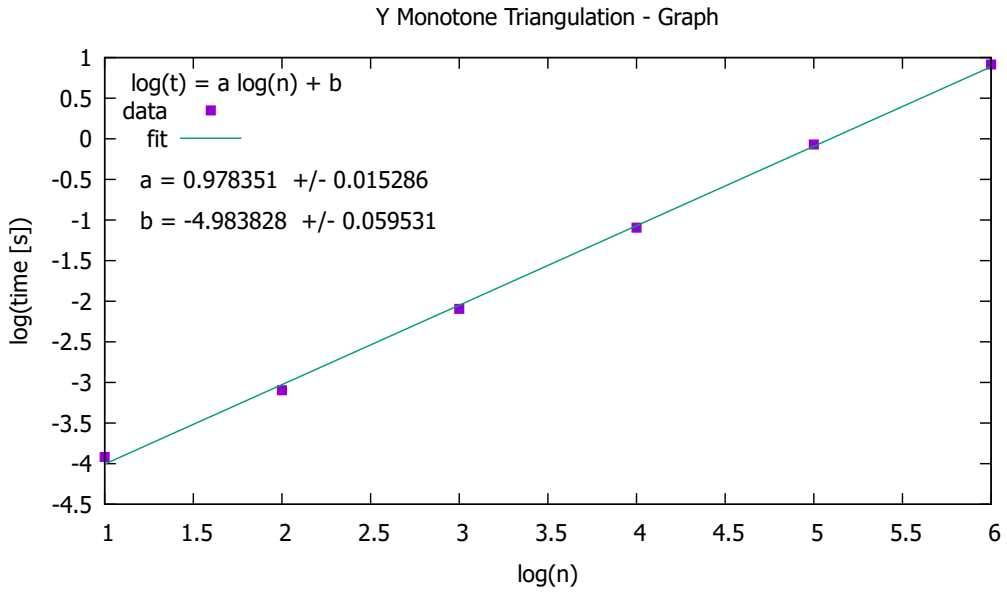
Do testu złożoności obliczeniowej algorytmu wyznaczania kierunków monotoniczności wykorzystany został generator wielokątów przypominających pas z ząbkowanymi brzegami, użyty również w testach algorytmu podziału monotonicznego wielokąta prostego. Test potwierdza, że praktyczna złożoność obliczeniowa implementacji wynosi $O(n)$ i zgadza się z teoretyczną złożonością algorytmu. Obrazuje to wykres A.11.



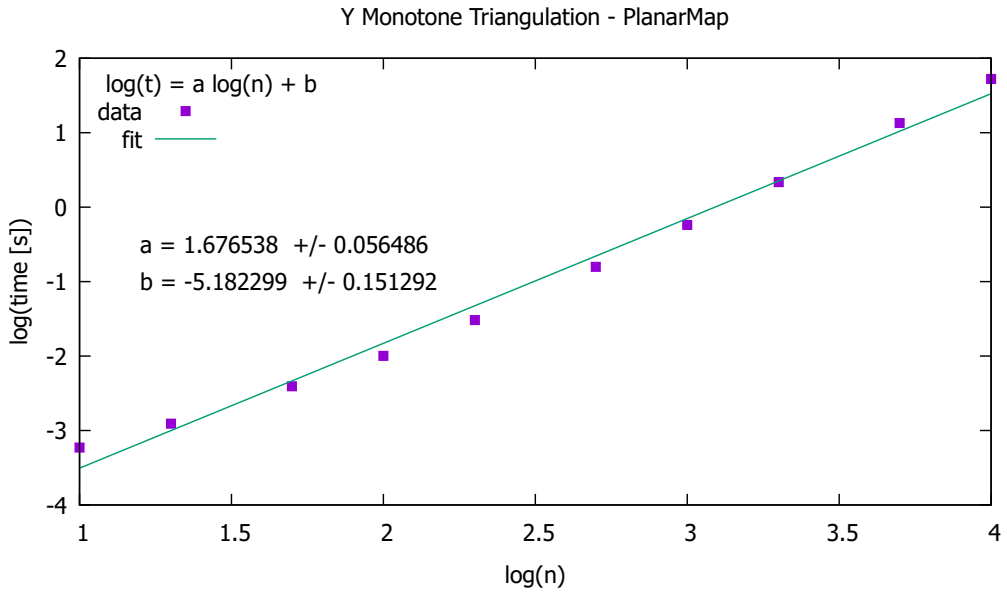
Rysunek A.1. Wykres wydajności algorytmu triangulacji wachlarzowej z wykorzystaniem klasy Graph do przechowywania danych wyjściowych. Współczynnik $a = 0.981(13)$ potwierdza praktyczną złożoność implementacji $O(n)$.



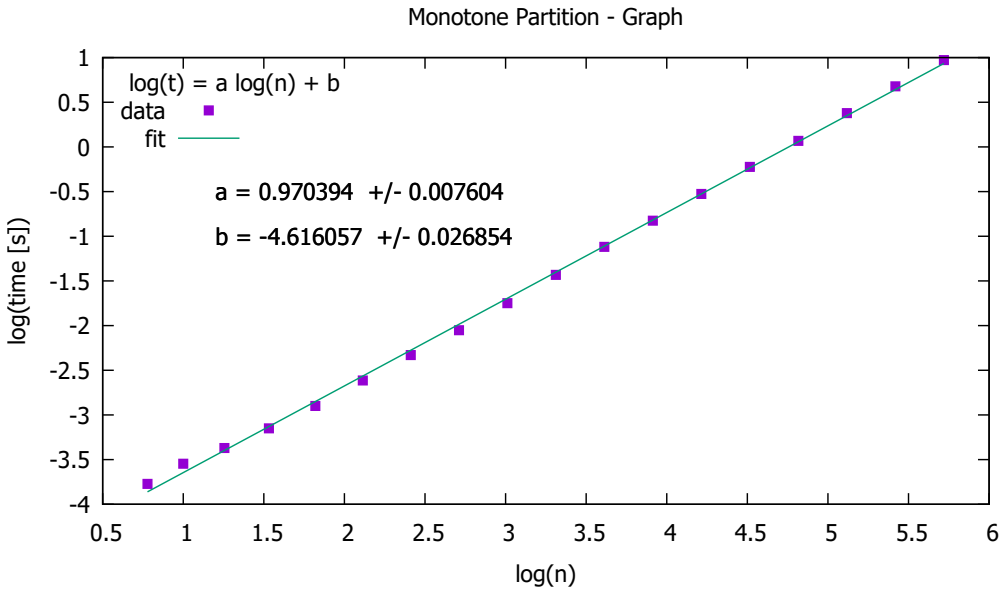
Rysunek A.2. Wykres wydajności algorytmu triangulacji wachlarzowej z wykorzystaniem klasy PlanarMap do przechowywania danych wyjściowych. Współczynnik $a = 1.831(31)$ potwierdza, że przechowywanie triangulacji w postaci mapy planarnej jest kosztowne obliczeniowo, przez co implementacja posiada znacznie gorszą złożoność obliczeniową niż teoretyczna złożoność $O(n)$.



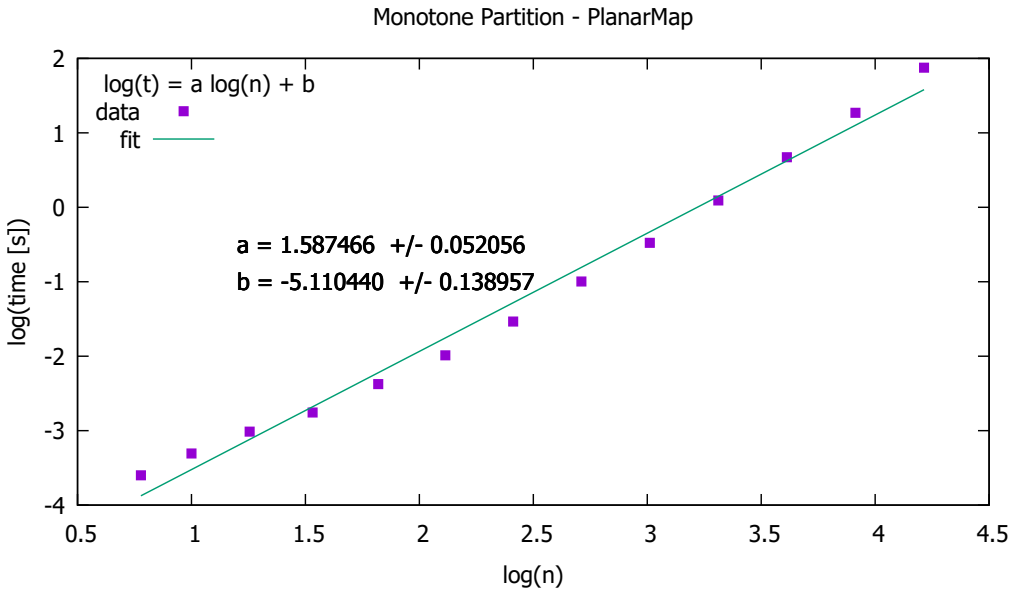
Rysunek A.3. Wykres wydajności algorytmu triangulacji wachlarzowej z wykorzystaniem klasy Graph do przechowywania danych wyjściowych. Współczynnik $a = 0.978(15)$ potwierdza praktyczną złożoność implementacji $O(n)$.



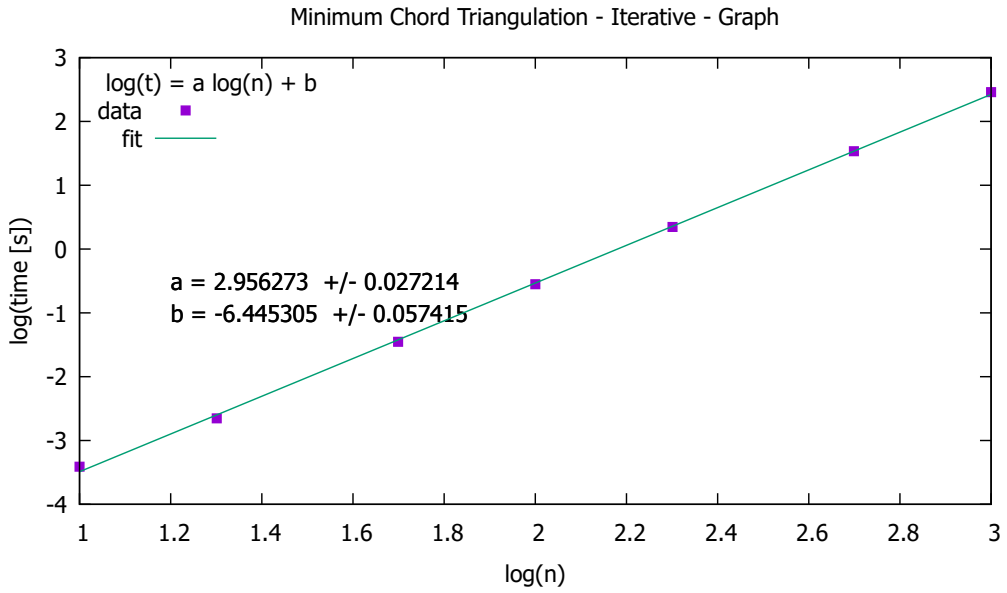
Rysunek A.4. Wykres wydajności algorytmu triangulacji wielokąta y-monotonicznego z wykorzystaniem klasy PlanarMap do przechowywania danych wyjściowych. Współczynnik $a = 1.677(56)$ potwierdza, że przechowywanie triangulacji w postaci mapy planarnej jest kosztowne obliczeniowo, przez co implementacja posiada znacznie gorszą złożoność obliczeniową niż teoretyczna złożoność $O(n)$.



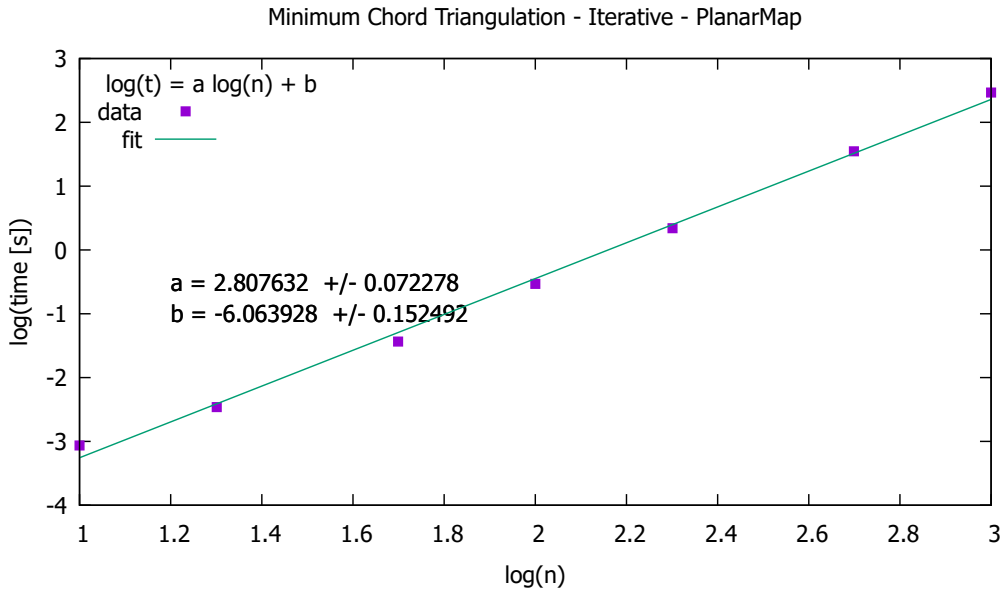
Rysunek A.5. Wykres wydajności algorytmu podziału monotonicznego wielokąta prostego z wykorzystaniem klasy Graph do przechowywania danych wyjściowych. Współczynnik $a = 0.970(8)$ potwierdza złożoność implementacji rzędu $O(n \log n)$, a praktycznie nawet $O(n)$ dla danego wielokąta.



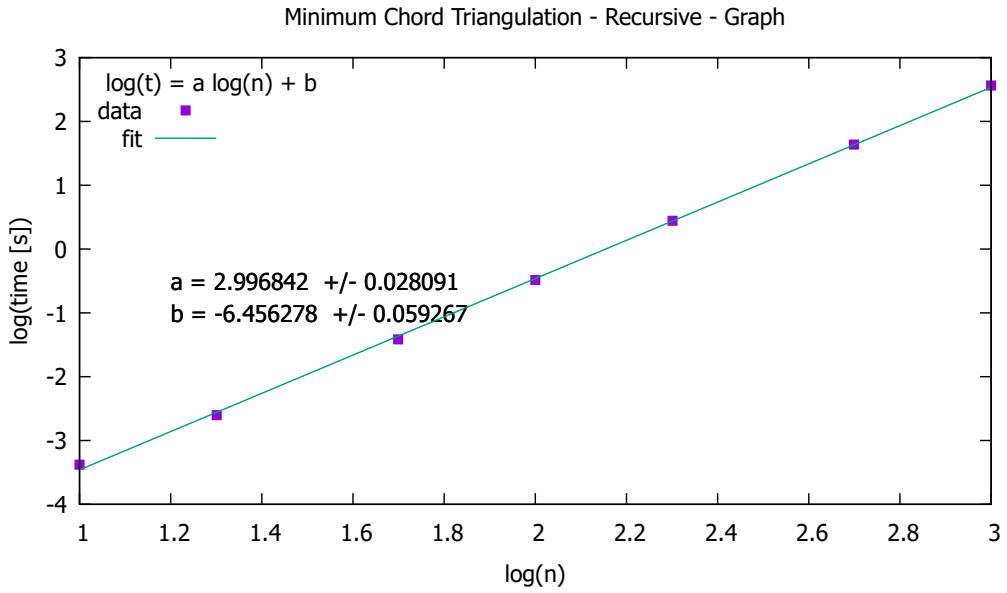
Rysunek A.6. Wykres wydajności algorytmu podziału monotonicznego wielokąta prostego z wykorzystaniem klasy PlanarMap do przechowywania danych wyjściowych. Współczynnik $a = 1.587(52)$ potwierdza, że przechowywanie triangulacji w postaci mapy planarnej jest kosztowne obliczeniowo, przez co implementacja posiada znacznie gorszą złożoność obliczeniową niż teoretyczna złożoność $O(n \log n)$.



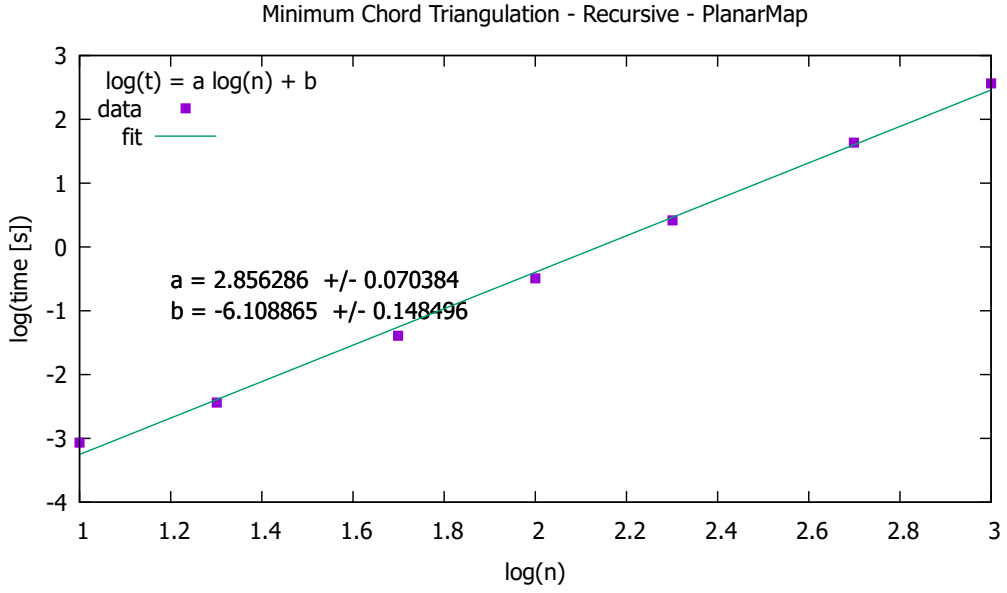
Rysunek A.7. Wykres wydajności algorytmu triangulacji z minimalizacją długości cięciw w wersji iteracyjnej z wykorzystaniem klasy Graph do przechowywania danych wyjściowych. Współczynnik $a = 2.956(27)$ potwierdza praktyczną złożoność implementacji $O(n^3)$.



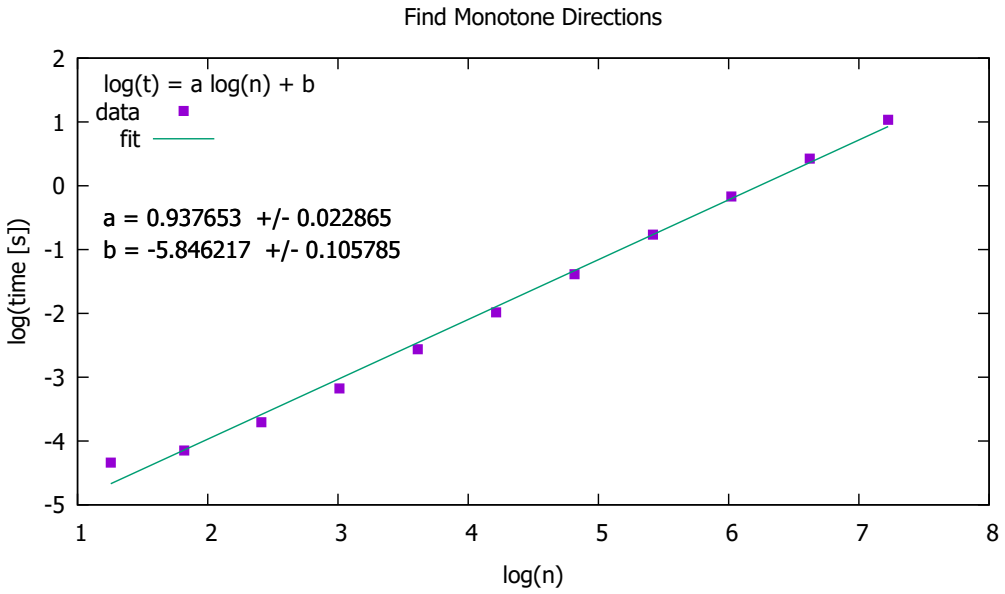
Rysunek A.8. Wykres wydajności algorytmu triangulacji z minimalizacją długości cięciw w wersji iteracyjnej z wykorzystaniem klasy PlanarMap do przechowywania danych wyjściowych. Współczynnik $a = 2.808(72)$ potwierdza praktyczną złożoność implementacji $O(n^3)$.



Rysunek A.9. Wykres wydajności algorytmu triangulacji z minimalizacją długości cięciw w wersji rekurencyjnej z wykorzystaniem klasy Graph do przechowywania danych wyjściowych. Współczynnik $a = 2.997(28)$ potwierdza praktyczną złożoność implementacji $O(n^3)$.



Rysunek A.10. Wykres wydajności algorytmu triangulacji z minimalizacją długości cięciw w wersji rekurencyjnej z wykorzystaniem klasy PlanarMap do przechowywania danych wyjściowych. Współczynnik $a = 2.856(70)$ potwierdza praktyczną złożoność implementacji $O(n^3)$.



Rysunek A.11. Wykres wydajności algorytmu wyznaczania kierunków monotoniczności. Współczynnik $a = 0.938(23)$ potwierdza praktyczną złożoność implementacji $O(n)$.

Bibliografia

[1] Wikipedia, Monotone polygon, 2024,
`https://en.wikipedia.org/wiki/Monotone_polygon`.

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

[3] Andrzej Kapanowski, planegeometry, GitHub repository, 2024,
`https://github.com/ufkapano/planegeometry/`.

[4] Wikipedia, Polygon partition, 2024,
`https://en.wikipedia.org/wiki/Polygon_partition`.

[5] Wikipedia, Polygon triangulation, 2024,
`https://en.wikipedia.org/wiki/Polygon_triangulation`.

[6] Gabriela Mazur, *Wielokąty monotoniczne w geometrii obliczeniowej*, Uniwersytet Jagielloński, Kraków 2021.

[7] Anna Sarnavska, *Nakładanie map w geometrii obliczeniowej*, Uniwersytet Jagielloński, Kraków 2020.

[8] Wikipedia, Polygon, 2024,
`https://en.wikipedia.org/wiki/Polygon`.

[9] Wikipedia, Simple polygon, 2024,
`https://en.wikipedia.org/wiki/Simple_polygon`.

[10] Wikipedia, Convex polygon, 2024,
`https://en.wikipedia.org/wiki/Convex_polygon`.

[11] Wikipedia, Concave polygon, 2024,
`https://en.wikipedia.org/wiki/Concave_polygon`.

[12] Wikipedia, Doubly connected edge list, 2024,
`https://en.wikipedia.org/wiki/Doubly_connected_edge_list`.

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

[14] Franco P. Preparata, Kenneth J. Supowit, *Testing a simple polygon for monotonicity*, Information Processing Letters 12 (4), 161-164 (1981).

[15] Python Docs, math, 2024,
`https://docs.python.org/3/library/math.html`.

[16] Daniel Jimenez, *Analysis of Algorithms. Lecture 12. Dynamic Programming continued. Triangulation of a Convex Polygon*, The University of Texas at San Antonio, 1998.
`https://www.cs.utexas.edu/~djimenez/utsa/cs3343/lecture12.html`

[17] Wikipedia, Minimum-weight triangulation, 2024,
`https://en.wikipedia.org/wiki/Minimum-weight_triangulation`.

[18] Python Docs, timeit, 2024,
`https://docs.python.org/3/library/timeit.html`.