

# Wyznaczanie otoczki wypukłej

## Quickhull

Szymon Tomaszewski

25/12/2024

### 1. Treść zadania

Wyznaczanie otoczki wypukłej, np. Graham, Jarvis, quickhull

[[https://en.wikipedia.org/wiki/Convex\\_hull](https://en.wikipedia.org/wiki/Convex_hull)].

### 2. Przedmowa

Dokument ten ma na celu przybliżenie działania programu od strony teoretycznej jak i technicznej, techniczne zagadnienia zostały bardziej poruszone w komentarzach znajdujących się w kodzie. Dokument został dodatkowo wzbogacony o ilustracje, znajdujące się w rozdziale "Załączniki", w celu lepszego zrozumienia działania programu.

### 3. Teoria

#### 3.1. Otoczka wypukła - wprowadzenie

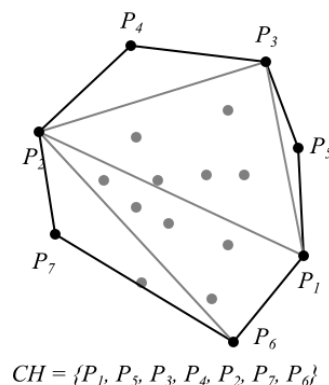
Otoczka wypukła jest to najmniejszy zbiór wypukły taki, że każdy element danego zbioru  $A$  zawiera się w tym wielokącie lub leży na jego brzegu. Można oczywiście rozważać przestrzenie wielowymiarowe lecz rozwiązanie takiego problemu jak i jego wizualizacja jest znacznie cięższa.

Należy także napomnieć czym jest zbiór wypukły, ponieważ pozwoli nam to na niewielką optymalizację w kodzie. Jest to podzbiór pewnej przestrzeni zawierający wraz z dowolnymi dwoma jego punktami odcinek je łączący. (*Załącznik 3*)

#### 3.2. Quickhull

Quickhull to algorytm dziel i zwyciężaj który wyznacza otoczkę wypukłą zbioru punktów umieszczonych w przestrzeni o dowolnej liczbie wymiarów. Został on odkryty przez Williama Eddy'ego i Alexa Bykata oraz Greena i Silvermana.

Średnia złożoność tego algorytmu wynosi  $O(n \log(n))$  a pesymistycznie  $O(n^2)$ . Najgorsza złożoność występuje jeśli wszystkie punkty należą do otoczki wypukłej a najlepsza gdy każdy wyznaczony punkt tworzy równy podział pozostałych punktów.



Źródło Wikipedia <https://pl.wikipedia.org/wiki/Quickhull>

### 3.2.1. Algorytm

Krok 1:

W zbiorze punktów znajdujemy dwa skrajne punkty (**A** i **B**) - minimalną i maksymalną współrzędną **x**.

Krok 2:

Dzielimy zbiór punktów na dwa podzbiory **S1** oraz **S2** które odpowiednio znajdują się nad prostą **AB** i pod nią.

Krok 3:

Następnie wywołujemy rekurencyjne program dla podanych podzbiorów.

Procedura ta przyjmuje trzy argumenty, punkty **A** i **B** oraz podzbiór **P** na którym ma być wykonany.

- Jeśli **P** jest puste to kończymy.
- Jeśli **P** ma jeden element to należy od do otoczki.
- W przeciwnym razie:
  - Szukamy punktu **F** najbardziej oddalonego od prostej **AB** - ten punkt należy do otoczki wypukłej. Wszystkie punkty wewnątrz trójkąta **ABF** odrzucamy.
  - Następnie znajdujemy zbiór **S1** punktów znajdujących się po prawej stronie prostej **AF** oraz analogiczny zbiór **S2** dla prostej **BF**.
  - Wywołujemy rekurencyjne program dla podanych podzbiorów.

### 3.3. Implementacja

Bardziej szczegółowe informacje dotyczące implementacji zostały zawarte w komentarzach znajdujących się w kodzie.

#### 3.3.1. Punkty na płaszczyźnie - 2D

Została stworzona struktura w pliku *Point.h*, która reprezentuje punkty na płaszczyźnie. Zaimplementowano także porównania dwóch punktów poprzez przeciążenie operatora "==".

#### 3.3.2. Odległość między punktami

Odległość jest liczona między prostą stworzoną przez punkty **A** i **B** a punktem obecnie sprawdzanym **P**.

Licznik to inaczej pole równoległoboku utworzonego przez wektory **AB** i **AP**, mianownik to długość wektora **AB** czyli odległość między punktami **A** i **B**. Mianownik jest wyliczany jest z twierdzenia Pitagorasa (*załącznik 1*)

### 3.3.3. Szukanie najmniejszego i największego elementu

Skorzystamy z `std::sort` z biblioteki `algorithm`.

`std::sort(begin, end, compare);`

- `begin` i `end` - określają zakres elementów do sortowania
- `compare` - jest to alternatywna funkcja która określa regułę porównywania algorytmów - w naszym przypadku jest to funkcja `compare`. Domyślnie jest wykorzystywany "<" lecz w naszym przypadku będziemy sortować elementy po x. Nasz algorytm sortujący musi zwracać `true` jeśli pierwszy argument jest mniejszy niż drugi i `false` w przeciwnym razie.

Z uwagi na to że wykorzystaliśmy `vector` możemy teraz skorzystać z `.front()` i `.back()`.

### 3.3.4. Iloczyn wektorowy

Funkcja `cross` odpowiada za obliczenie iloczynu wektorowego która jest wykorzystywana w funkcji `getSides`.

### 3.3.5. Określanie po której stronie znajdują się linie

Wykorzystywany jest tutaj wynik iloczynu wektorowego dla każdego punktu.

Jeśli wynik jest równy 0 to punkt P leży na linii AB, jeśli wynik jest większy od 0 to punkt leży po lewej stronie linii AB a w przeciwnym przypadku po prawej.

### 3.3.6. Najdalszy punkt od linii AB

Dla każdego punktu obliczamy odległość od linii AB za pomocą funkcji `distance`, jeśli punkt znajduje się dalej niż najdalszy dotychczasowy punkt to go aktualizujemy.

### 3.3.7. Właściwy algorytm - quickHull

Już na starcie możemy zauważyć, że jeśli zbiór złożony jest z mniej niż 3 punktów nie da się stworzyć otoczki wypukłej, według definicji, którą podaliśmy na początku. Następnie znajdujemy najbardziej oddalone punkty (A i B) od siebie (względem osi x) i mamy pewność, że należą one do otoczki więc je dodajemy. Następnie dzielimy punkty na lewe i prawe względem odcinka wyznaczonego przez punkty A i B.

W kolejnym kroku wykorzystujemy funkcję pomocniczą. W niej określamy najdalej odległy punkt F (który należy do otoczki) i dzielimy zbiór na elementy które są poza trójkątem ABF, czyli leżą po lewej stronie linii AF i FB. (Załącznik 2)

Funkcja pomocnicza jest wykonywana rekurencyjnie aż zabraknie punktów.

### 3.3.8. Wykorzystane biblioteki

#### 3.3.8.1. `algorithm`

Wykorzystywane w celu zastosowania sortowania programu i przy testowaniu.

#### 3.3.8.2. `cmath`

Wykorzystana w celu obliczania wartości bezwzględnej w odległości.

#### 3.3.8.3. `vector`

Aby ułatwić operacje na punktach.

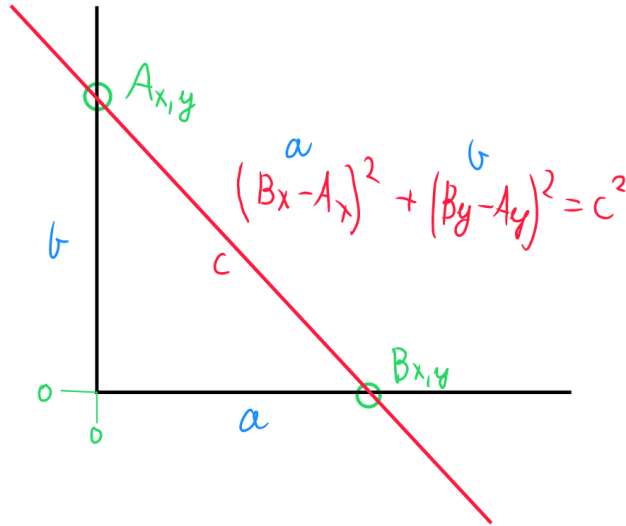
#### 3.3.8.4. `iostream`

Wyświetlanie danych.

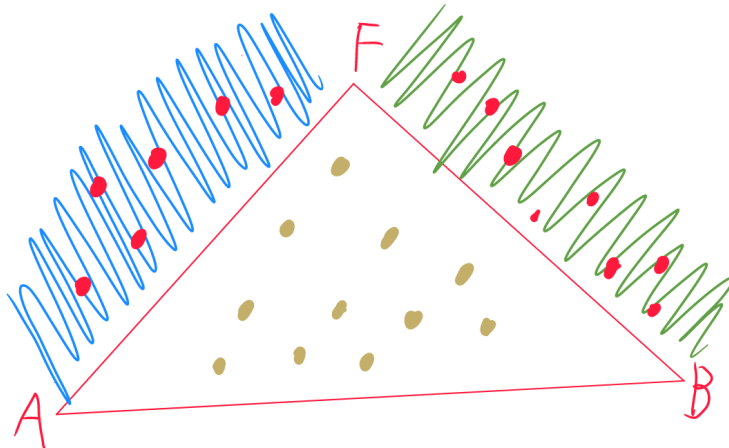
### 3.4. Testowanie

Testowanie jest realizowane w pliku `test.cpp`. Polega ono na sprawdzeniu czy dany program zwróci taki sam zbiór punktów co oczekiwany. Została stworzona struktura `TestCase` do łatwiejszego rozbudowywania testów.

### 3.5. Załączniki

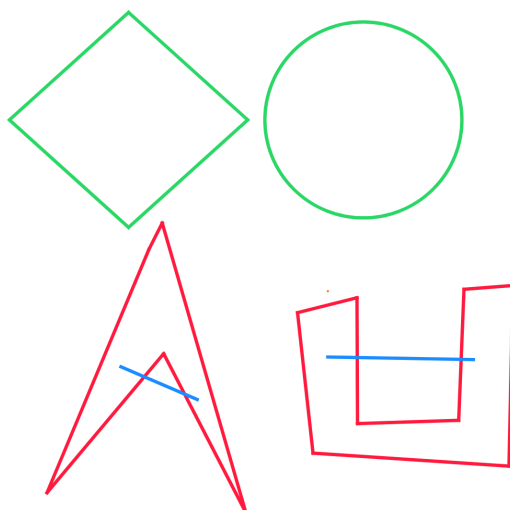


(Załącznik 1) Wyliczenia z twierdzenia Pitagorasa



(Załącznik 2) - Wyznaczanie kolejnych punktów do sprawdzenia (czerwone kropki na ilustracji) przez uwzględnianie tylko lewej strony AF (kolor niebieski) i FB (kolor zielony).

Kropki wewnątrz utworzonej figury są pomijane



(Załącznik 3) Na zielono zaznaczono poprawne zbiory wypukłe a na czerwono zaznaczono niepoprawne wraz z uzasadnieniem.

### 3.6. Źródła

- 1) <https://pl.wikipedia.org/wiki/Quickhull>
- 2) [https://pl.wikipedia.org/wiki/Zbi%C3%B3r\\_wypuk%C5%82y](https://pl.wikipedia.org/wiki/Zbi%C3%B3r_wypuk%C5%82y)
- 3) <https://www.educative.io/answers/what-is-the-quick-hull-algorithm>