

[AAL] Convex hull

Piotr Frątczak
(300207)

Bartosz Świtalski
(300279)

24 stycznia 2021

1 Problem

Dany jest zbiór punktów w przestrzeni trójwymiarowej. Wyznaczyć zadaną dokładnością jego powłokę wypukłą (tzn. tak, aby żaden z punktów nie wystawał ponad znalezioną powłokę bardziej niż zostanie to dopuszczone). Porównać czas obliczeń i wyniki różnych metod.

2 Decyzje projektowe

Wybrany język programowania to C++. Do projektu zostanie dołączony plik `doc/manual.pdf` z instrukcją uruchamiania programu.

Zakładamy, że nie wiemy, które z podanych punktów są punktami powierzchniowymi¹.

Dokładność

Zbiór punktów na wejściu programu zostaje przetworzony na odpowiednie struktury danych przy użyciu algorytmu wokselizacji przy ustalonej dokładności. Dokładność decyduje o liczbie podprzestrzeni (wokseli), na które zostanie podzielona chmura otrzymanych punktów. Następnie na podstawie otrzymanych wokseli tworzony jest zbiór punktów reprezentujący wejściową chmurę punktów.

Dane testowe

Testy w pierwszej części zostaną wykonane na losowo wygenerowanych danych.

Testy w drugiej części zostaną wykonane na danych wygenerowanych pod kątem sprawdzenia konkretnych przypadków testowych (np. punkty w prostopadłościanie, w sferze, w pobliżu sfery).

¹leżą na powierzchni ściany powłoki wypukłej

3 Przetwarzanie wstępne

Powłoka zewnętrzna dla podanego na wejściu programu zbioru punktów w przestrzeni zostaje wyznaczona z ustaloną dokładnością d . Wyznaczony zostaje prostopadłościan ograniczający chmurę punktów, jego wymiary zależą od maksymalnych współrzędnych punktów z chmury. Wspomniany prostopadłościan zostaje podzielony na sześciany o wymiarach $d \times d \times d$. Następnie po kolei przeglądany jest każdy punkt p_w z wejściowej chmury punktów i sprawdzany jest sześcian s , w którym ten punkt się znajduje. Jeżeli s został wcześniej odwiedzony, przechodzimy do kolejnego punktu z chmury. W przeciwnym wypadku oznaczamy s jako odwiedzony oraz dodajemy nowy punkt p_s do zbioru punktów reprezentujących chmurę punktów w programie. Współrzędne p_s są równe środkowi ciężkości sześcianu s .

Pseudokod

Algorithm 1 Algorytm przetwarzania wstępnego

```
1: procedure PREPROCESSING(punkty,  $d$ )
2:    $w_x \leftarrow \max_x(\textit{punkty}) - \min_x(\textit{punkty})$ 
3:    $w_y \leftarrow \max_y(\textit{punkty}) - \min_y(\textit{punkty})$ 
4:    $w_z \leftarrow \max_z(\textit{punkty}) - \min_z(\textit{punkty})$ 
5:    $l_x \leftarrow \lceil w_x/d \rceil$   $\triangleright$  wyznaczenie liczby sześcianów dla każdego wymiaru BBox
6:    $l_y \leftarrow \lceil w_y/d \rceil$ 
7:    $l_z \leftarrow \lceil w_z/d \rceil$ 
8:   bool odwiedzone $[l_x][l_y][l_z] \leftarrow$  nowa tablica z wartościami false
9:   for all  $p \in \textit{punkty}$  do
10:     $i_x \leftarrow (p.x - \textit{BBox.min.x})/d$   $\triangleright$  dzielenie całkowitoliczbowe
11:     $i_y \leftarrow (p.y - \textit{BBox.min.y})/d$ 
12:     $i_z \leftarrow (p.z - \textit{BBox.min.z})/d$ 
13:    if not odwiedzone $[i_x][i_y][i_z]$  then
14:      odwiedzone $[i_x][i_y][i_z] \leftarrow$  true
15:      dodaj nowy obiekt reprezentujący punkt o współrzędnych równych
16:      środkowi ciężkości sześcianu  $[i_x][i_y][i_z]$ 
17:    end if
18:  end for
19: end procedure
```

W ten sposób liczba punktów, na których zostają wykonane właściwe algorytmy szukania powłoki zewnętrznej zostaje zmniejszona, dzięki czemu zwiększona zostaje prędkość działania programu, ponieważ złożoność algorytmu przetwarzania wstępnego jest równa $O(n)$.

4 Implementowane algorytmy

Opis

- **Algorytm naiwny** Naiwne sprawdzanie każdej trójki punktów czy jest ścianą zewnętrzną powłoki. Wykonają się 4 pętle, każda po n razy.
- **Gift Wrapping** Wykorzystuje fakt, że znamy już punkty należące do powłoki wypukłej i , mając krawędź k należącą do takiej ściany, znajdziemy kolejną ścianę przez znalezienie punktu po drugiej stronie krawędzi niż znana już ściana. Dzięki temu nie musimy przeglądać wszystkich trójek, lecz dobieramy kolejny punkt do krawędzi znanej ściany zewnętrznej tak, aby nowa ściana także była zewnętrzna.
- **Incremental** Polega na dołączaniu punktów leżących na powłoce do znanych już punktów leżących na powłoce.

Działanie

Algorithm 2 Algorytm naiwny

```
1: procedure NAIVE(punkty)
2:   for all  $i \in \textit{punkty}$  do
3:     for all  $j \in \textit{punkty} \setminus \{i\}$  do
4:       for all  $k \in \textit{punkty} \setminus \{i, j\}$  do
5:         dodaj  $(i, j, k)$  do zbioru ścian powłoki zewnętrznej
6:         for all  $l \in \textit{punkty} \setminus \{i, j, k\}$  do
7:           if  $l$  jest po zewnętrznej stronie płaszczyzny opisanej przez  $(i, j, k)$  then
8:             usuń  $(i, j, k)$  ze zbioru ścian powłoki zewnętrznej
9:           end if
10:        end for
11:      end for
12:    end for
13:  end for
14:  return zbiór ścian powłoki zewnętrznej
15: end procedure
```

Algorithm 3 Gift Wrapping

```
1: procedure GIFTWRAPPING(punkty)
2:   while nie znaleziono pierwszej ściany do
3:     algorytmNaiwny(punkty)           ▷ do momentu znalezienia pierwszej ściany
4:   end while
5:   for all krawędź  $k \in$  ściany ze zbioru  $s$  ścian powłoki zewnętrznej do
6:     if  $k$  jest jedną z krawędzi dokładnie jednej ściany w  $s$  then
7:       for all  $p \in$  punkty do
8:         if  $p \notin$  żadnej ściany ze zbioru ścian powłoki zewnętrznej then
9:           ▷ ściślej:  $p \notin$  żadnej ściany powłoki zewnętrznej
10:          if  $(k, p)$  opisuje ścianę zewnętrzną then
11:            dodaj ścianę  $(k, p)$  do  $s$ 
12:          end if
13:        end if
14:      end for
15:    end if
16:  end for
17: end procedure
```

Algorithm 4 Incremental

```
1: procedure INCREMENTAL(punkty)
2:   znajdź pierwsze 4 punkty leżące na powłoce zgodnie z algorytmem Gift Wrapping
3:   for  $i \leftarrow 5$ , sizeof(punkty) do
4:     for all  $s \in$  zbioru ścian powłoki zewnętrznej do
5:       if  $p[i]$  leży po zewnętrznej stronie  $s$  then
6:         połącz  $p[i]$  krawędziami z powłoką zewnętrzną
7:       else
8:         odrzuć  $p[i]$ 
9:       end if
10:    end for
11:  end for
12: end procedure
```

Teoretyczna pesymistyczna złożoność

Algorytm	$O(T(n))$
Algorytm naiwny	$O(n^4)$
Gift Wrapping	$O(n^2)$
Incremental	$O(n^2)$

5 Użycie

Dane z pliku wejściowego

```
./chull -in [plik_wejscowy] -a [algorytm]
```

Automatyczna generacja danych na podstawie parametrów:

```
./chull -n [liczba_punktow] -d [dokladnosc]  
-seed [seed] -out [plik_wyjsciowy] -a [algorytm]
```

Testowanie z generacją danych, pomiarem czasu i prezentacją wyników

```
./chull -seed [seed] -n [liczba_punktow]  
-p [liczba_problemov] -step [krok]  
-r [liczba_instancji_problemu] -a [algorytm]
```

6 Konwencje

Wejście

Parametry `-n`, `-p`, `-step`, `-r`, `-seed` powinny być liczbami całkowitymi. Parametr `-d` powinien być liczbą rzeczywistą z kropką, jako separatorem dziesiętnym.

Każdy wiersz pliku wejściowego składa się z 3 liczb rzeczywistych opisujących współrzędne punktu na osiach x, y, z , oddzielonych średnikiem, z kropką jako separatorem dziesiętnym. Wiersze kończą się znakiem nowej linii. Kolejność punktów jest dowolna.

Przykładowe wejście:

```
./chull -seed 1234 -n 1000 -p 3 -step 500 -r 5 -a incremental
```

Pomiar czasu i prezentacja wyników dla 3 problemów o wielkościach 1000, 1500 i 2000. Dla każdej wielkości losowanych (`seed = 1234`) 5 instancji problemu. Doświadczenie zostanie przeprowadzone dla algorytmu `incremental`. Wyniki zostaną zapisane do folderu

Przykładowy plik wejściowy `input.txt`:

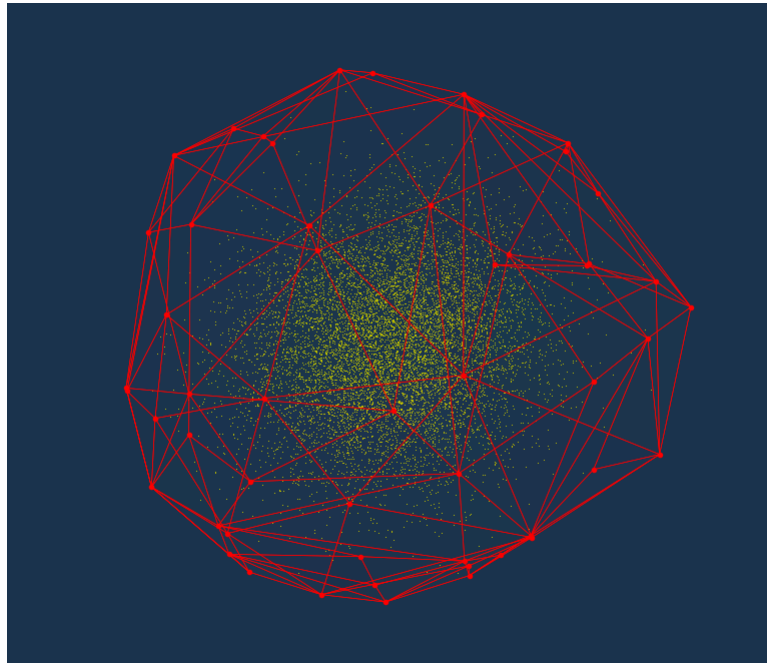
```
2.15;0.333;4.01  
4.665;2.643;9.06  
1;1;1 2;10;2
```

Wyjście

Dla dwóch pierwszych opcji uruchomienia wynikowa otoczka wypukła jest zapisywana do plików (`points.data`, `vertices.data`, `faces.data`). Dla testowania z generacją danych, pomiarem czasu i prezentacją wyników, dane są zapisywane w pliku `times.data` w folderze `/convex-hull/times`.

Reprezentacja graficzna

Dla danych z pliku wejściowego oraz automatycznej generacji na podstawie parametrów można wygenerować wizualizację.



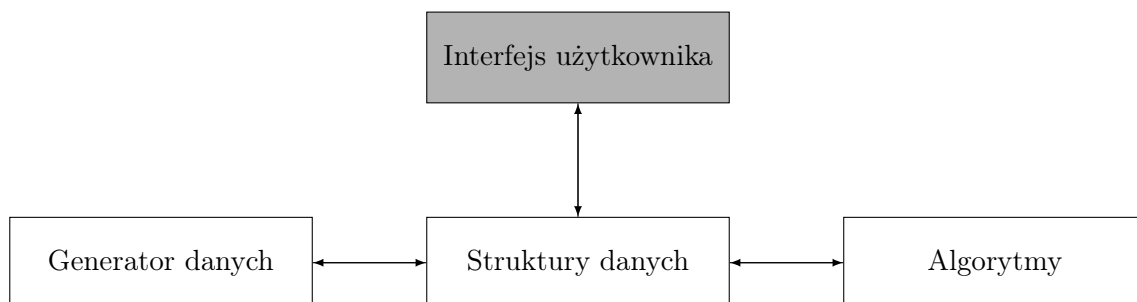
Rysunek 1: Przykładowa wizualizacja dla $n=10000$ punktów

Aby wygenerować wizualizację, należy uruchomić skrypt `plot.py`:

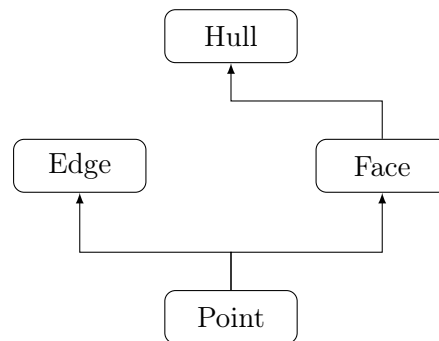
```
/convex-hull$  
  
cd /convex-hull/visuals  
python3 plot.py
```

7 Struktura programu

Podział na moduły



Hierarchia klas



8 Testowanie

Głównym celem testowania było sprawdzenie i porównanie złożoności czasowych różnych algorytmów. W tym celu porównane zostały czasy działania algorytmów na różnych wielkościach problemu.

Aby odtworzyć poniższe wyniki należy wykonać program z użyciem ziarna dla generatora danych równego 96879. Czas wykonania został uśredniony dla 10 wykonania.

9 Wyniki

Oznaczenia

- n - rozmiar problemu (liczba punktów)
- $t(n)$ - średni czas wykonania 5 uruchomień dla wartości n
- $q(n) = t(n) \cdot T(n_{\text{mediana}}) / (T(n) \cdot t(n_{\text{mediana}}))$

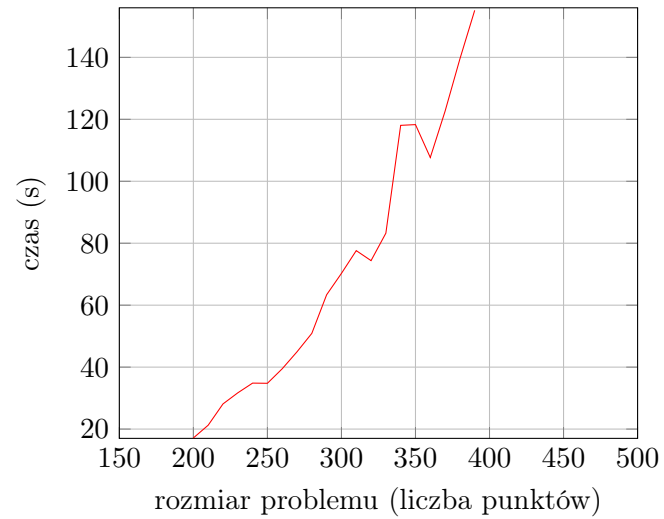
Zestawienie tabelaryczne

Algorytm naiwny $O(n^4)$		
n	t(n) [s]	q(n)
200	17.12	1.23
210	21.25	1.26
220	28.12	1.38
230	31.70	1.31
240	34.84	1.21
250	34.77	1.03
260	39.47	1.00
270	44.94	0.98
280	50.89	0.95
290	63.36	1.03
300	70.26	1.00
310	77.60	0.97
320	74.38	0.82
330	83.24	0.81
340	118.05	1.02
350	118.29	0.91
360	107.67	0.74
370	122.63	0.75
380	139.49	0.77
390	155.18	0.77

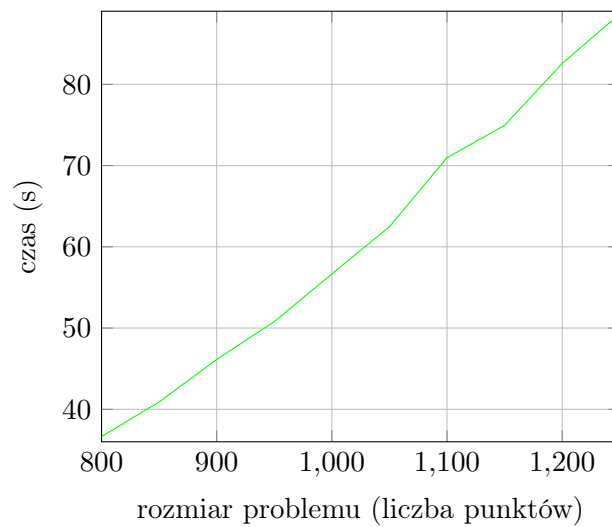
Algorytm giftWrapping $O(n^2)$		
800	36.64	1.01
850	40.90	1.00
900	46.12	1.01
950	50.79	0.99
1000	56.65	1.00
1050	62.46	1.00
1100	70.96	1.04
1150	74.94	1.00
1200	82.57	1.01
1250	88.66	1.00

Algorytm incremental $O(n^2)$		
30000	1.29	12.12
130000	6.44	3.22
230000	11.88	1.90
330000	17.13	1.33
430000	22.71	1.04
530000	33.26	1.00
630000	37.68	0.80
730000	41.42	0.66
830000	46.91	0.58
930000	53.78	0.53

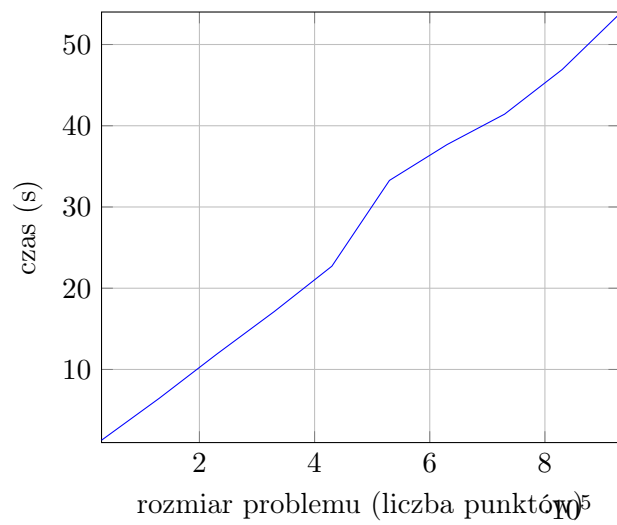
Zestawienie na wykresie



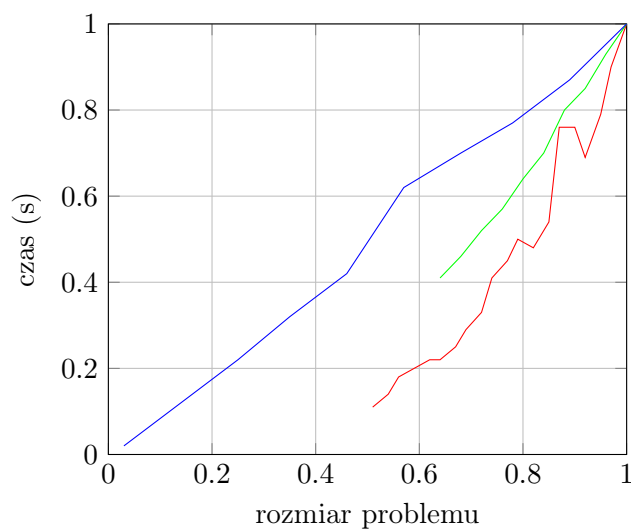
Rysunek 2: Czas działania algorytmu **naive** w funkcji rozmiaru problemu



Rysunek 3: Czas działania algorytmu **giftWrapping** w funkcji rozmiaru problemu



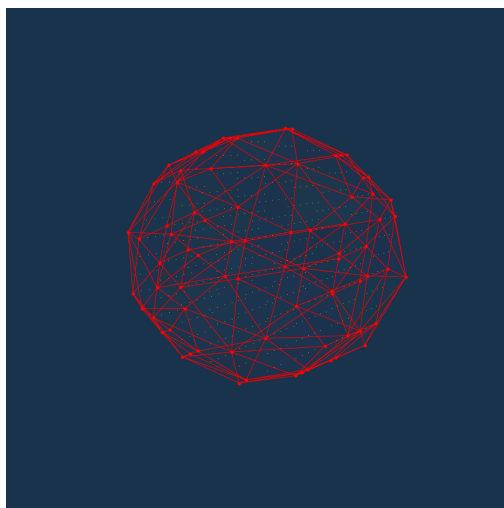
Rysunek 4: Czas działania algorytmu **incremental** w funkcji rozmiaru problemu



Rysunek 5: Czas działania implementowanych algorytmów w funkcji rozmiaru problemu w postaci znormalizowanej

Testowanie spreparowanych danych

Algorytmy były testowane dla danych reprezentujących chmurę punktów w kształcie sfery oraz sześcianu.



Rysunek 6: Przykładowa wizualizacja chmury punktów reprezentujących sferę.

Dla danych reprezentujących sześcián algorytmy działają gorzej. Generalnie algorytmy działają gorzej, jeśli mają do czynienia z dużą liczbą punktów na jednej płaszczyźnie.

10 Wnioski

Te same problemy można rozwiązać o wiele szybciej korzystając z różnych algorytmów. Niektóre algorytmy bazują na zauważeniu pewnej zależności pomagającej wielokrotnie przyspieszyć asymptotyczny czas jego działania. W taki sposób algorytm **GiftWrapping** opiera się na sprawdzaniu wszystkich możliwych ścian, jednak jest sprawniejszy od **naiwnego**, ponieważ wykorzystuje już zebraną wiedzę o ścianach otoczki. Inne algorytmy wynikają z zupełnie innego podejścia, jak **incremental**, który dołącza punkty do już istniejącej otoczki.

W przypadku algorytmu **naiwnego** działał on szybciej niż przewidywano. Pomimo prostej i jednoznacznej do oszacowania złożoności algorytmu **naiwnego**, wyniki testów wykazały tendencję spadkową wartości $q(n)$. Algorytm polega na czterokrotnie zagnieżdżonym przejrzeniu wektora wszystkich punktów, stąd złożoność $O(T(n)) = n^4$, jednak wyniki obliczeń wskazują na przeszacowanie jego złożoności. Taki rezultat może wynikać z np. z optymalizacji programu przez kompilator.

Zgodnie z obliczonymi wartościami współczynnika zgodności oceny teoretycznej z pomiarem czasu $q(n)$ algorytm **GiftWrapping** został precyzyjnie oszacowany, ponieważ wartości współczynnika są równe, bądź bardzo bliskie 1.0. Dla tego algorytmu potwierdzają się oczekiwania,

ponieważ algorytm sprawdza każdy punkt dla każdej pary spośród trzech punktów tworzących ścianę otoczki, co można sprowadzić do podwójnie zagnieżdżonej pętli po ilości punktów, stąd złożoność $O(T(n)) = n^2$.


Według przeprowadzonych doświadczeń, algorytm **incremental** został przeszacowany jeszcze bardziej niż **naiwny**. Jego oczekiwana złożoność wynosi $O(T(n)) = n^2$. Lecz w praktyce w pętli iterującej przez wszystkie otrzymane na wejściu punkty, te punkty, które znajdują się w środku aktualnie znalezionej otoczki (otoczka jest rozbudowywana z przebiegiem algorytmu) są pomijane, stąd szybsze wykonanie algorytmu.

Porównując wyniki wykonania algorytmów **incremental** i **GiftWrapping**, zauważalna jest różnica w wielkości problemów dla porównywalnych czasów wykonania, mimo że oba algorytmy mają taką samą złożoność czasową. Algorytm **GiftWrapping** wykonuje się prawie 60s dla problemów rzędu 1000 punktów, podczas gdy **incremental** w takim czasie rozwiązuje problemy o 3 rzędy wielkości większe. Jest tak ponieważ notacja O pomija stałe przy ocenie złożoności, dla rozmiarów problemów dążących do nieskończoności czasu wykonania będą dążyć do tej samej wartości.

11 Podsumowanie

Złożony projekt wprowadzający w tematykę analizy algorytmów. Dzięki implementacji kilku algorytmów do wyliczania otoczki wypukłej chmury punktów w przestrzeni trójwymiarowej poznano podstawy technik przeprowadzania analizy złożoności algorytmu.

12 Powiązane linki

 [Repozytorium projektowe](#)