Geometria obliczeniowa

Otoczka wypukła dla zbioru punktów w przestrzeni dwuwymiarowej

Wojciech Dymek
Informatyka
semestr 7
2016/2017

1. Cel projektu

Celem projektu była implementacja oraz wizualizacja trzech algorytmów, nie omawianych na zajęciach laboratoryjnych, tworzenia otoczki wypukłej w przestrzeni dwuwymiarowej:

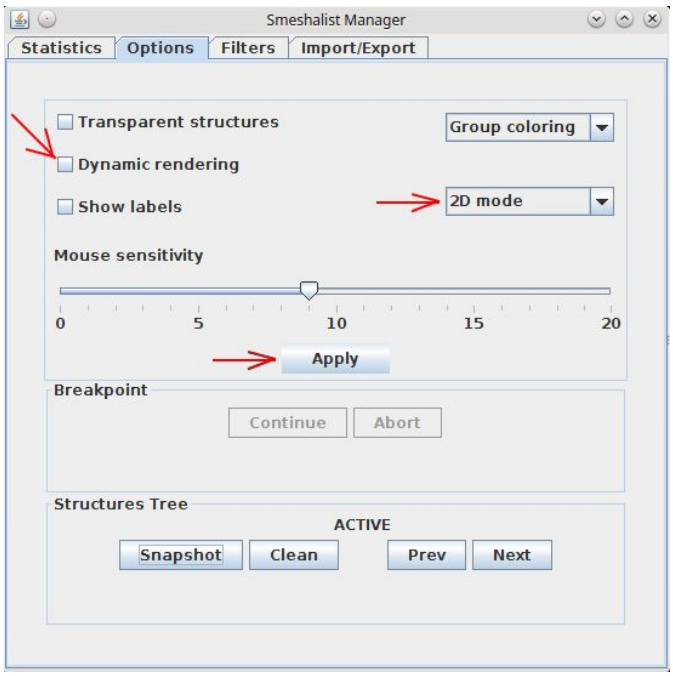
- algorytmu dziel i rządź
- algorytmu QuickHull
- algorytmu Chan'a

2. Wizualizacja algorytmów

W celu wizualizacji kolejnych kroków algorytmu została wykorzystana współtworzona praca inżynierska. Przewodnik oraz sposób użycia zostały opisane w dokumentacji dostępnej pod adresem: https://github.com/CharlieBr/Smeshalist/tree/master/Dokumentacja

Aplikacja została dołączona wraz z projektem. W celu zbudowania aplikacji Smeshalist. należy wykonać polecenie *make all* lub *make build*. Wygenerowany zostanie plik wykonywalny *Smeshalist*. Po jego uruchomieniu powinny się pojawić dwa okna - główne okno *Smeshalist* służące do wizualizacji oraz *Smeshalist Manager*.

UWAGA w celu poprawnej wizualizacji, należy w zakładce *Options* odznaczyć opcję *Dynamic rendering*, wybrać *2D mode* oraz potwierdzić wybór przyciskiem *Apply*.



Aplikacja wymaga zainstalowania biblioteki *freeGLUT* - `sudo apt-get install freeglut3-dev`.

Projekt został wykonany w dwóch wariantach - do celów wizualizacji (wymaga powyższej aplikacji) oraz do celów badania poprawności i czasu działania.
Do projektu zostały dołączone gotowe skrypty uruchomieniowe dla każdego z algorytmów, wraz z dwoma przykładowymi zestawami danych.
W celu uruchomienia projektu w trybie wizualizacji, należy przejść proces instalacji aplikacji Smeshalist, uruchomić ją, a następnie uruchomić odpowiedni skrypt - nazwy odpowiadają poszczególnym zaimplementowanym algorytmom. Aby przejść do kolejnego kroku algorytmu, należy wcisnąć przycisk *Continue* w zakładce *Options* w okienku *SmeshalistManager'a*. Cześć algorytmów pozwala na przejście krokowo z góry określonej liczby iteracji, później zostaje przełączony w tryb automatycznej animacji.
W celu uruchomienia projektu bez wizualizacji, należy dodać argument *noVis* w momencie uruchamiania skryptów.
Argumenty uruchomieniowe:

- *data* - ścieżka, pod którą znajduje się plik z danymi, w odpowiednim formacie (każda linijka zawiera współrzędne punktu oddzielone spacją)
- *alg* - określa typ algorytmu (CHAN | DAC - DivideAndConquer | GRAHAM | JARVIS | QH - QuickHull)
- *maxSubCHsize* - maksymalny rozmiar sub-otoczek - dotyczy algorytmów Chan'a i QuickHull
- *subCHgenerator* - typ generatora sub-otoczek - dotyczy tylko algorytmu Chan'a.
  1 - losowe przypisanie punktów do podzbiorów
  2 - podział wejściowego zbioru na pseudo-klastry - wykorzystywane do celów wizualizacyjnych

3. Implementacja algorytmów

Opis algorytmów opiera się na implementacji bez części wizualizacyjnej.

3.1. Algorytm dziel i rządź

Pakiet *divideandconquer*.

Implementacja została rozbita na dwa osobne moduły - właściwą implementację algorytmu w klasie *DivideAndConquer.java* oraz *TangentService.java*, która zawiera logikę do badania czy zadana prosta stanowi styczną do otoczki.
Badanie styczności polega na sprawdzeniu znaku wyznacznika bezpośrednich sąsiadów punktu.

Klasa *DivideAndConquer.java* zawiera pozostała logikę potrzebną do wyznaczenia wyjściowej otoczki. Pierwszym krokiem jest podział wejściowego zbioru punktów na podzbiory zadanej wielkości oraz wyznaczenie sub-otoczek przy pomocy algorytmu Grahama. Odpowiada za to metoda *computeCHInternal*. W przypadku odpowiednio małego zbioru wejściowego wyznacza ona sub-otoczki, bądź też wylicza medianę na podstawie x-owej współrzędnej punktów, dokonuje podziału i rekurencyjnie wywołuje samą siebie dla obu podziałów. Wynikiem działania jest lista sub-otoczek, posortowana rosnąco na podstawie x-owej współrzędnej.

Główna część algorytmu zawiera się w metodzie *mergeSubCHs* oraz *mergeTwoCH*. Pierwsza z nich pobiera listę sub-otoczek i odpowiada za prawidłową kolejność ich sklejania, metoda *mergeTwoCH* zawiera zaś logikę potrzebną do wykonania tego procesu i zwrócenia pojedynczej, sklejonej otoczki.

Pseudokod metody *mergeTwoCH* (dla wyszukiwania górnej otoczki):

A, ***B*** - wejściowe sub-otoczki, posortowane rosnąco względem x-owej współrzędnej
$\boldsymbol{EX_A}$ <- wierzchołek sub-otoczki ***A***, leżący maksymalnie na prawo
$\boldsymbol{EX_B}$ <- wierzchołek sub-otoczki ***B***, leżący maksymalnie na lewo

dopóki *(odcinek* $\boldsymbol{EX_AEX_B}$ *nie stanowi górnej stycznej do otoczek* ***A*** *oraz* ***B****) lub*
(odcinek $\boldsymbol{EX_AEX_{prev(B)}}$ *stanowi górną styczną do otoczek* ***A*** *oraz* ***B****) lub*

*(odcinek $EX_{next(A)}EX_B$ stanowi górną styczną do otoczek **A** oraz **B**):*

***dopóki** (odcinek EX_AEX_B nie stanowi górnej stycznej do otoczki **B**) lub*
*(odcinek $EX_AEX_{prev(B)}$ stanowi górną styczną do otoczki **B**):*

EX_B <- $EX_{prev(B)}$

***dopóki** (odcinek EX_AEX_B nie stanowi górnej stycznej do otoczki **A**) lub*
*(odcinek $EX_{next(A)}EX_B$ stanowi górną styczną do otoczki **A**):*

EX_A <- $EX_{next(A)}$

Po przejściu przez powyższy zestaw pętli, zmienne EX_A oraz EX_B wyznaczają górną styczną do obu otoczek. Analogicznie wyznaczana jest styczna dolna.
Ostatnim krokiem jest stworzenie otoczki wynikowej. Powstaje ona poprzez sklejenie ciągu wierzchołków leżących na odpowiednich łańcuchach ograniczonych przez wyznaczone wierzchołki stanowiące styczne do otoczek.

Pseudokod:

upperA <- *wierzchołek na otoczce **A**, będący jednym z końców **górnej** stycznej do otoczek **A** oraz **B***
upperB <- *wierzchołek na otoczce **B**, będący jednym z końców **górnej** stycznej do otoczek **A** oraz **B***
downA <- *wierzchołek na otoczce **A**, będący jednym z końców **dolne** stycznej do otoczek **A** oraz **B***
downB <- *wierzchołek na otoczce **B**, będący jednym z końców **dolnej** stycznej do otoczek **A** oraz **B***

r <- ***upperA***
rób: *dodaj wierzchołek **r** do wynikowej otoczki;* ***r*** <- ***next(r)***
dopóki: r != ***downA***

r <- ***downB***
rób: *dodaj wierzchołek **r** do wynikowej otoczki;* ***r*** <- ***prev(r)***
dopóki: r != ***upperB***

3.2. Algorytm QuickHull

Pakiet *quickhull*.

Implementacja algorytmu zawiera się w klasie *QuickHull.java*.

Pierwszym krokiem działania algorytmu jest wyznaczenie punktów skrajnych względem współrzędnej x - logika zawiera się w metodzie *getInitialPoints(List)*. Wyznaczone punkty na pewno stanowią wierzchołki otoczki oraz rozdzielają wejściowy zbiór punktów na dwa podzbiory, na których zostanie uruchomiony właściwy algorytm. Podział dokonywany jest na podstawie wartości wyznacznika. Wynikiem działania głównej metody *computeCH* jest lista wierzchołków otoczki, zbudowana jako odpowiednia konkatenacja wyznaczonych rekurencyjnie wierzchołków oraz wierzchołków skrajnych. Pseudokod:

a <- punkt leżący maksymalnie na lewo
b <- punkt leżący maksymalnie na prawo
A <- zbiór punktów leżących powyżej odcinka ***ab***
B <- zbiór punktów leżących poniżej odcinka ***ab***

zwróć *computeCH(**a**, **b**, **A**)* ∪ {***a***} ∪ *computeCH(**b**, **a**, **B**)* ∪ {***b***}

Funkcja *computeCH(Vertex, Vertex, List*) zawiera główną logikę algorytmu. Pierwszym krokiem jest wyznaczenie punktu leżącego najdalej od zadanego odcinka. Odbywa się to w metodzie *getFarthestPoint*, samo wyszukiwanie najdalszego punktu odbywa się poprzez wyliczenie pola trójkąta wyznaczonego na odcinku i analizowanym punkcie. Z racji tego, że długość odcinka jest stała, najbardziej oddalonemu punktowi będzie odpowiadać trójkąt o największym polu.
Po wyznaczeniu tego punktu następuje podział wejściowego zbioru i rekurencyjne wywołanie na wygenerowanych podzbiorach. Pseudokod:

points - wejściowy zbiór punktów
a, b - punkty stanowiące końce odcinka podziału zbioru nadrzędnego
A, B - zbiory punktów leżących po lewej stronie odcinka odpowiednio ***ac*** oraz ***cb***

c <- *getFarthestPoint(**points**)*
dla każdego p należącego do points:
    jeżeli p *leży po lewej stronie odcinka* ***ac***:
        dodaj ***p*** *do zbioru* ***A***

***jeżeli p** leży po lewej stronie odcinka **cb**:*

*dodaj **p** do zbioru **B***

***zwróć** computeCH(**a**, **c**, **A**)* ∪ {**c**} ∪ *computeCH(**c**, **b**, **B**)*

3.3. Algorytm Chan'a

Pakiet *chan*.

Implementacja zawiera się w klasie *Chan.java*.

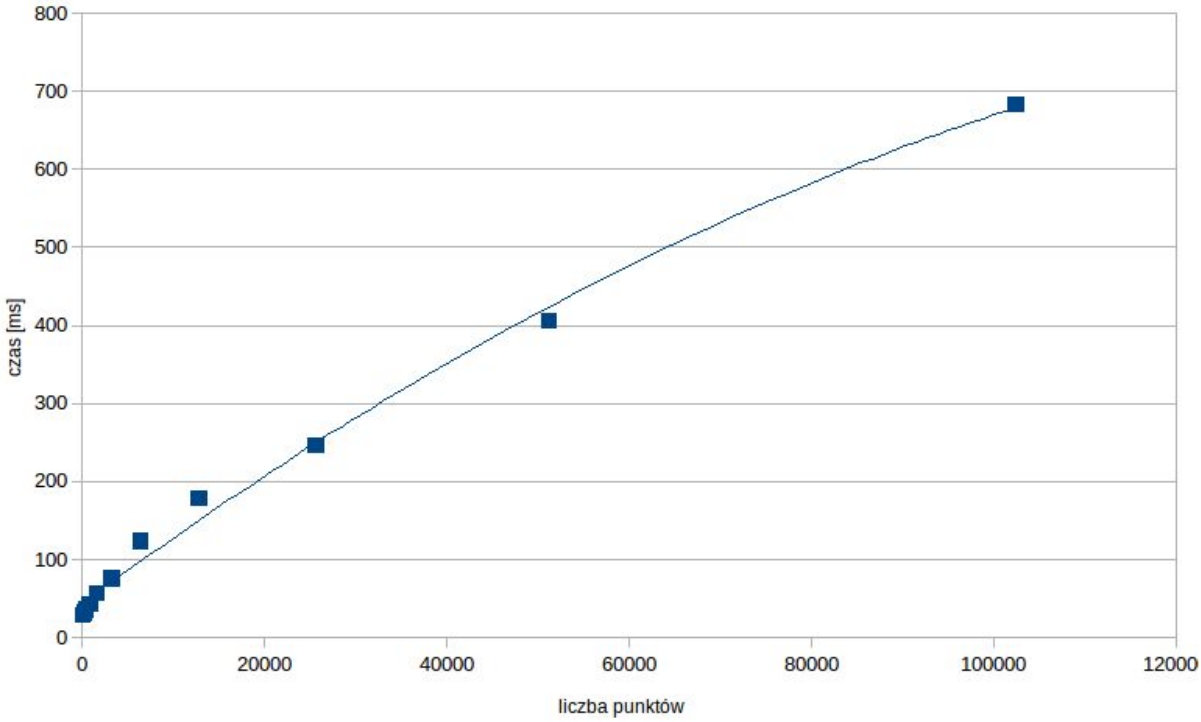
Pierwszym krokiem algorytmu jest podział zbioru wejściowego na podzbiory o zadanej wielkości. W zależności od przekazanego parametru, odbywa się to w dwojaki sposób - punkty są przydzielane do kolejnych otoczek na podstawie ich indeksu w liście wejściowej, bądź też (na potrzeby wizualizacji) wyznaczane są pseudo-klastry - logika wyznaczania podzbiorów zawiera się w metodach odpowiednio *generateSubSets* oraz *generateSubSets2*.

Kolejnym krokiem jest wyznaczenie otoczek dla podzbiorów. Do tego celu używany jest ponownie algorytm Graham'a. Dla otrzymanych sub-otoczek uruchamiamy algorytm Jarvis'a. Kluczowym elementem tego algorytmu jest wybór kandydatów na kolejny wierzchołek wynikowej otoczki. Zbiór kandydatów tworzony jest jako zbiór prawych stycznych do tych otoczek (względem aktualnie rozważanego wierzchołka otoczki wynikowej) oraz kolejny wierzchołek sub-otoczki, na której aktualnie się znajdujemy. Wyszukiwanie prawych stycznych został zaimplementowany na podstawie kodu załączonego pod poniższym linkiem:
http://tomswitzer.net/2010/12/2d-convex-hulls-chans-algorithm/
Logika pozwalająca na wyszukiwanie prawych stycznych znajduje się w metodzie *getCHCandidate*. Po wyznaczeniu zbioru kandydatów wystarczy wybrać wierzchołek tworzący największy kąt z ostatnio dodaną krawędzią otocki - będzie on stanowił kolejny wierzchołek wynikowej otoczki.

4. Badanie złożoności algorytmów

Prezentowane wykresy przedstawiają zależności pomiędzy ilością punktów w wejściowym zbiorze, a czasem działania algorytmu (podanym w ms).

4.1. Algorytm dziel i rządź



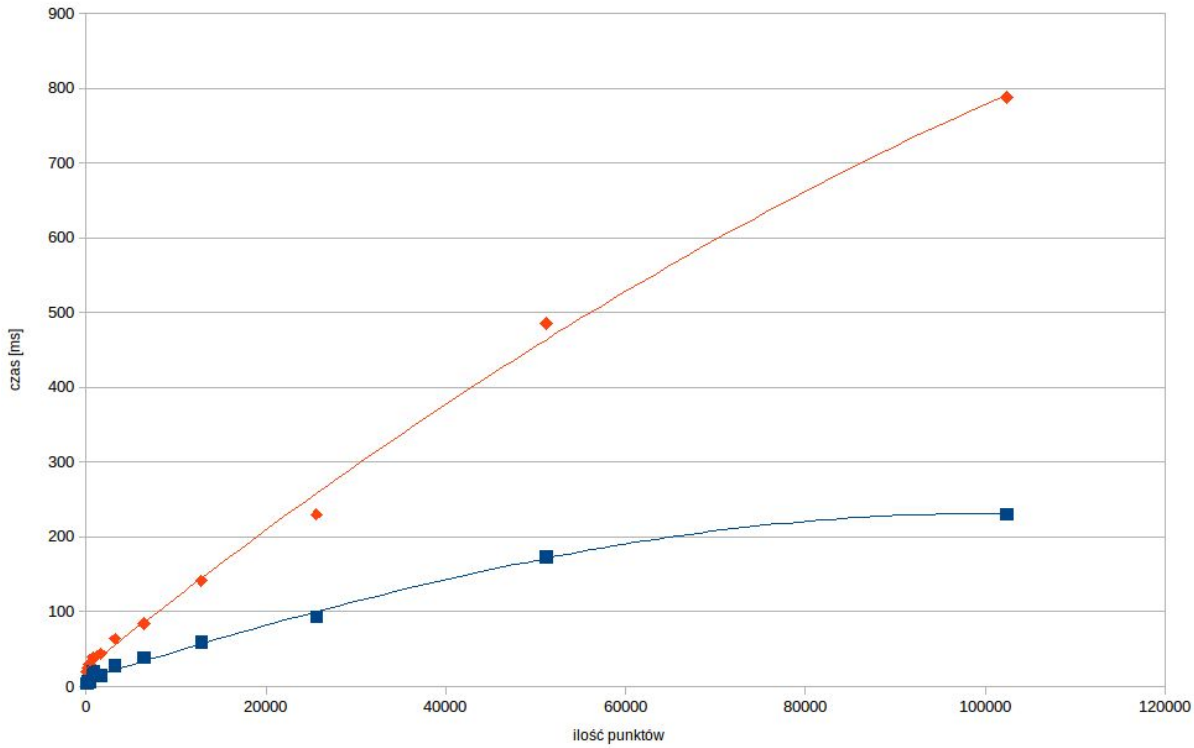
Dla celów wyznaczenia czasów działania, maksymalną wielkość podzbiorów wyliczano jako pierwiastek z mocy zbioru wejściowego.

W trakcie badania złożoności zauważyłem, że bardzo ważnym aspektem wpływającym na czas działania, była gęstość rozłożenia punktów wzdłuż osi x.

Dla większych gęstości czas działania algorytmu znacznie się wydłużał.

Wykres prezentuje średnie uzyskane czasy dla punktów leżących wewnątrz kwadratu o boku długości 2000j. Wyznaczona krzywa regresji sugeruje uzyskanie oczekiwanej złożoności $O(n\log n)$.

4.2. Algorytm QuickHull



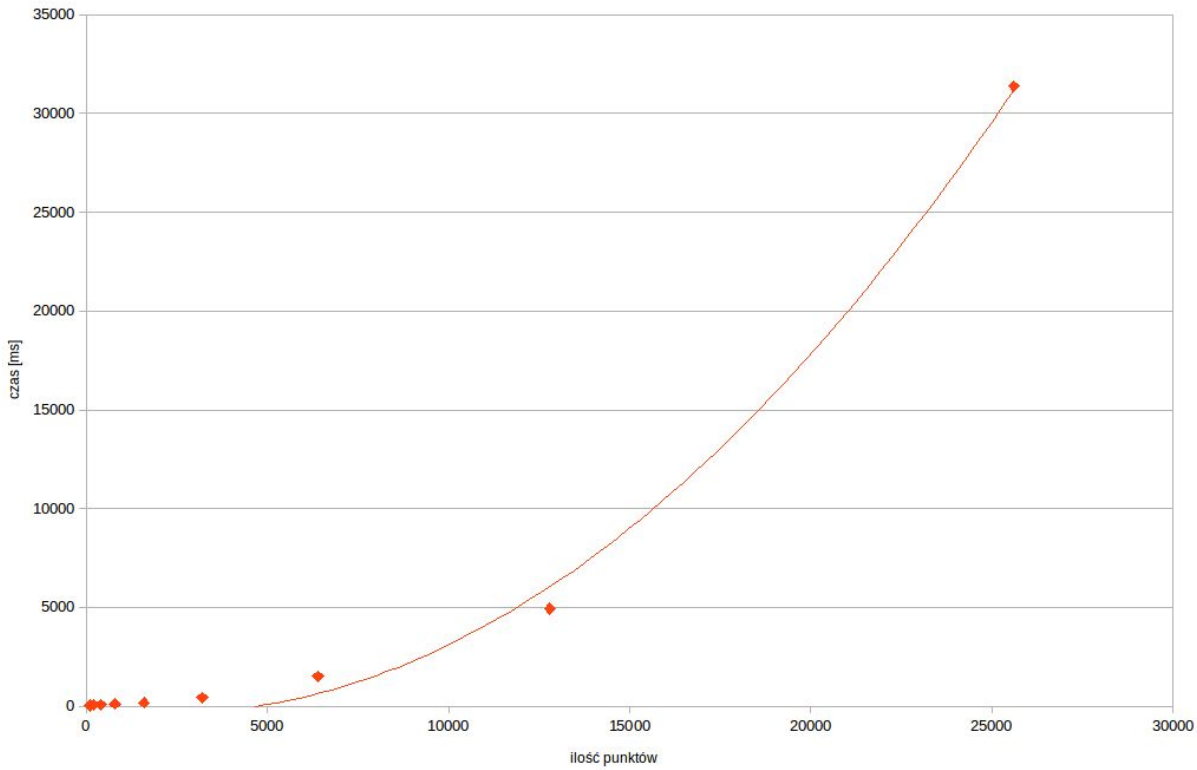
Złożoność algorytmu QuickHull została zbadana dla dwóch zbiorów danych:

1) losowe punkty leżące wewnątrz kwadratu o boku 2000j – kolor niebieski
2) losowe punkty leżące na okręgu o promieniu 1000j – kolor pomarańczowy

Dla (1) uzyskano oczekiwaną średnią złożoność $O(n \log n)$.

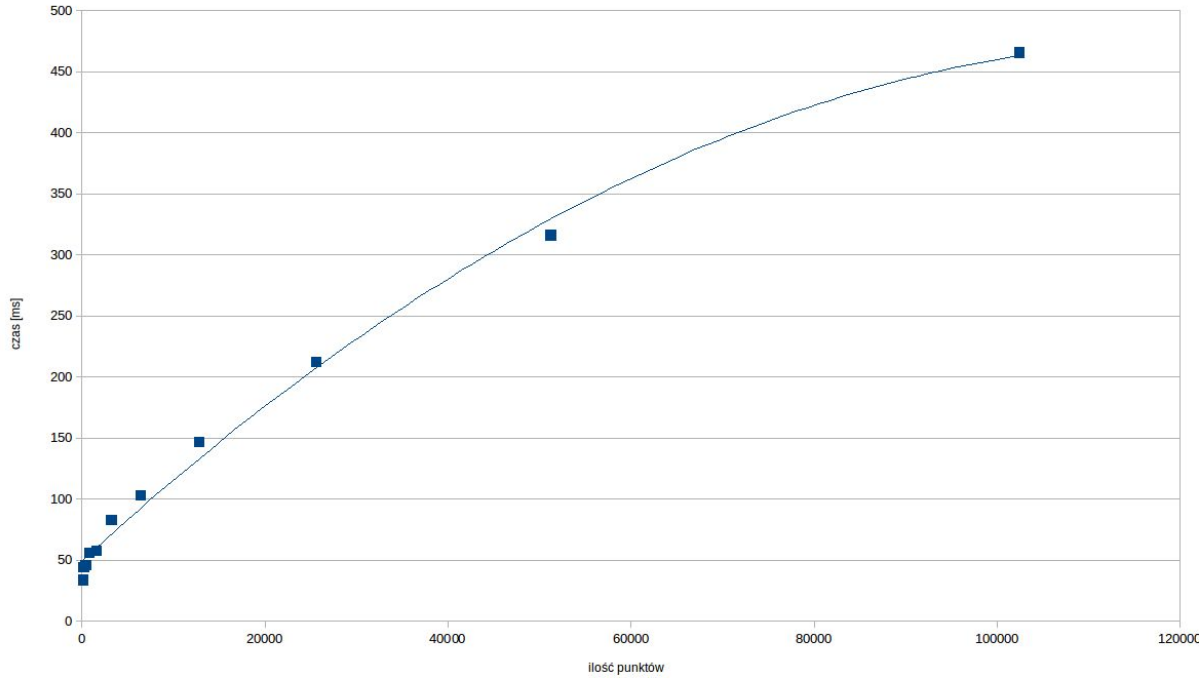
Dla punktów (2), które stanowią pesymistyczny przypadek dla algorytmu *QuickHull*, uzyskano złożoność pesymistyczną $O(n^2)$. Jednak na podstawie wyznaczonej krzywej regresji oraz bardzo małemu współczynnikowi dla stopnia drugiego, można wysunąć hipotezę, że pomimo złożoności kwadratowej, algorytm ten będzie osiągał czasy znacznie lepsze niż algorytmy Jarvisa w przypadku pesymistycznym.

Dla porównania, wykres czasu działania od ilości punktów dla algorytmu Jarvisa:



Dla liczności zbioru wejściowego równego 25'600, algorytm Jarvisa okazał się 337 razy wolniejszy od algorytmu QuickHull.

4.3. Algorytm Chan’a



Wyznaczona krzywa regresji sugeruje poprawność działania algorytmu oraz uzyskanie oczekiwanej złożoności $O(n \log k)$. Autor algorytmu podaje szereg optymalizacji, jednak ze względu na poziom ich skomplikowania, nie wszystkie zostały zaimplementowane. Sądzę, że dodanie kolejnych znacząco wpłynęłoby na spłaszczenie wykresu i poprawę czasu działania.

5. Bibliografia

1. https://pl.wikipedia.org/wiki/Quickhull – opis koncepcji oraz podstawowej wersji algorytmu QuickHull (dostęp 3.12.2016)

2. http://geomalgorithms.com/a15-_tangents.html – algorytm znajdowania stycznych do otoczek metodą binary search (dostęp 3.12.2016)

3. http://tomswitzer.net/2010/12/2d-convex-hulls-chans-algorithm/ - algorytm oraz przykładowa implementacja znajdowania stycznych do otoczek metodą binary search (dostęp 3.12.2016)